An Analysis of Primality Testing and Its Use in Cryptographic Applications

Jake Massimo

Thesis submitted to the University of London
for the degree of Doctor of Philosophy

Information Security Group
Department of Information Security
Royal Holloway, University of London

2020

Declaration

These doctoral studies were conducted under the supervision of Prof. Kenneth G. Paterson.

The work presented in this thesis is the result of original research carried out by myself, in collaboration with others, whilst enrolled in the Department of Mathematics as a candidate for the degree of Doctor of Philosophy. This work has not been submitted for any other degree or award in any other university or educational establishment.

Jake Massimo
April, 2020

Abstract

Due to their fundamental utility within cryptography, prime numbers must be easy to both recognise and generate. For this, we depend upon primality testing. Both used as a tool to validate prime parameters, or as part of the algorithm used to generate random prime numbers, primality tests are found near universally within a cryptographer's tool-kit. In this thesis, we study in depth primality tests and their use in cryptographic applications.

We first provide a systematic analysis of the implementation landscape of primality testing within cryptographic libraries and mathematical software. We then demonstrate how these tests perform under adversarial conditions, where the numbers being tested are not generated randomly, but instead by a possibly malicious party. We show that many of the libraries studied provide primality tests that are not prepared for testing on adversarial input, and therefore can declare composite numbers as being prime with a high probability. We also demonstrate that for a number of libraries, including Apple's CommonCrypto, we are able to construct composites that *always* pass the supplied primality tests.

We then explore the implications of these security failures in applications, focusing on the construction of malicious Diffie-Hellman parameters. These malicious parameter sets target the public key parameter validation functions found in these same cryptographic libraries – particularly within the ones that offer TLS implementations. Using the analysis performed on these library's primality tests, we are able to construct malicious parameter sets for both finite-field and elliptic curve Diffie-Hellman that pass validation testing with some probability, but are designed such that the Discrete Logarithm Problem (DLP) is relatively easy to solve. We give an application of these malicious parameter sets to OpenSSL and password authenticated key exchange (PAKE) protocols.

Finally, we address the shortcomings uncovered in primality testing under adversarial conditions by the introduction of a performant primality test that provides strong security guarantees across all use cases, while providing the simplest possible API. We examine different options for the core of our test, describing four different candidate primality tests and analysing them theoretically and experimentally. We then evaluate the performance of the chosen test in the use case of prime generation and discuss how our proposed test was fully adopted by the developers of OpenSSL through a new API and primality test scheduled for release in OpenSSL 3.0 (2020).

Acknowledgements

I would like to first thank my supervisor Kenny Paterson for his patience, motivation, and guidance throughout the course of my Ph.D. Kenny has provided me so much insight and development into all aspects of my academic work, and I thank him for always making time for this. I would also like to thank Martin Albrecht for his support as a mentor and collaborator, as well as Liz Quaglia.

I would like to thank Juraj Somorovsky and Steven Galbraith for giving me the opportunity to work with them as collaborators and to learn from their expertise. It was inspirational to see the level at which they work.

I am grateful to the Engineering and Physical Sciences Research Council (EPSRC) for funding my work as part of the Centre for Doctoral Training (CDT) at Royal Holloway, University of London. I also give thanks to my fellow CDT students, and in particular Joanne Woodage, for truly being a role model and a friend to me – I learned from you how to deal with all the things that can't be learned from supervision and was constantly motivated by your outstanding level of work. I would like also to thank the CDT leadership and support staff at Royal Holloway, particularly Carlos Cid and Claire Hudson.

I would also like to thank Matthew Campagna and Shay Gueron for hosting me at Amazon Web Services as an intern. Your team gave me an incredible opportunity to learn and work with so many other great researchers and engineers.

Lastly, I give thanks to my Mum, Dad, Sister, and Anna for your love, support, and for watching my presentations even when you don't understand them.

Publications

This thesis is based on the following three publications:

1. Martin R. Albrecht, Jake Massimo, Kenneth G. Paterson, and Juraj Somorovsky. Prime and Prejudice: Primality Testing Under Adversarial Conditions. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, Canada, October 15-19, 2018, pp. 281-298, ACM.* [5]

   This work originated with discussions between Martin and Kenny. I then joined the project as work began to flesh out these ideas. The main theoretical parts of the work arose from discussions between Martin, Kenny and me. The practical analysis of cryptographic libraries and mathematical software was in the main performed by me (and supported by all other authors), with the exception of the Bouncy Castle and Botan libraries, which were jointly worked on with Juraj Somorovsky.

2. Steven Galbraith, Jake Massimo, and Kenneth G. Paterson. Safety in Numbers: On the Need for Robust Diffie-Hellman Parameter Validation. In *IACR International Workshop on Public Key Cryptography, PKC 2019, Beijing, China, April 14-17, 2019, pp. 379-407, Springer.* [57]

   This work started as a continuation of the previous work between Kenny and me. All sections were therefore authored jointly between Kenny and me, with the exception of the theoretical work on the elliptic curve setting which was due to Steven. Steven then contributed to the authorship as a whole during the submission process.

3. Jake Massimo and Kenneth G. Paterson. A Performant, Misuse-Resistant API for Primality Testing. To appear in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security, CCS 2020, Orlando, USA, November 9-13, 2020, ACM.* [100]

   This work was contributed to equally by Kenny and me.

Contents

Chapter 1

Introduction

Contents

This chapter gives an overview of the thesis. We provide the motivation for our research and describe the contributions of this thesis. We also present the overall structure of the thesis.

1.1 Motivation

It is a truth universally acknowledged, that a single man in possession of a good fortune, must be in want of a prime [11]. Prime numbers are often referred to as the building blocks of the natural numbers, but with more recent advancements in the field of cryptography, we see just how integral primes are to our everyday lives. With the rising prominence of the Internet, cryptography - the art of writing or solving codes for secret communication - has evolved into a daily practice for most. Supported by a deep and fast-growing research field, prime numbers and their use in cryptography have a new found importance from the very first introduction of public-key cryptography [48], to cutting edge post-quantum systems such as isogeny-based cryptography [78].

The ubiquity of prime numbers throughout cryptography ensures that one is never too far away from relying upon them. For cryptographic schemes in which security arises from the extreme difficulty of certain computations, for example the factorisation of integers into primes or the discrete logarithm problem, the onus of the security of the scheme can fall onto the prime parameter itself. We see examples of such usage in public-key cryptography, with protocols like RSA and Diffie-Hellman

key exchange. We participate in cryptographic protocols like these each time we use mobile messaging applications, make an online purchase, connect to a website encrypted with TLS, or when using smart cards through contactless payments and even government-issued IDs. Since many of these devices are low-power (or computationally light) we often look to find efficient methods of prime generation on the device, or source the prime parameters from elsewhere – be it an external server or even from a standard. This however opens up the possibility of vulnerability, either by an attack from a malicious party or from weak or erroneous implementation.

A failure in the selection of prime parameters can be disastrous. In the case of RSA, there are numerous examples of large scale implementations of parameter generation that have led to the compromise of the system's security as a whole. For example, RSA smart card moduli were generated using a predictable underlying structure which therefore could be reverse engineered to recover the private key factors by Coppersmith's algorithm [114]. As another example, a faulty pseudorandom number generator (PRNG) was used to generate RSA moduli for use in digital certificates, and led to the trivial factorisation of public keys that shared a common factor [73]. Similarly, we see that poor choices of primes taken from standards for use in Diffie-Hellman admit the use of small subgroup attacks [156]. We have even seen composite numbers believed to be prime being used in the Diffie-Hellman parameters used by a command-line data transfer utility `socat` [137].

A fundamental tool in a cryptographic developer's tool-kit is the primality test. This is an algorithm used to classify a given number as either composite or prime. Primality tests are implemented near universally throughout every cryptographic library or computer algebra system (in mathematical software). These primality tests provide utility in two main forms. The first is part of a fundamental step in prime generation. Since many cryptographic protocols require fresh prime parameters, a library must provide the ability to generate them as and when needed. There are numerous ways of performing prime generation, but a popular technique – due to it producing uniformly random primes in a fairly efficient manner, is to simply generate random integers of the desired bit length and test them for primality using a primality test. The second significant use case of primality testing in this setting is its use in validation and checking functions. These, primarily found in cryptographic libraries that offer TLS implementations (for example OpenSSL, Botan, Bouncy Castle, and

NSS) are functions that take as input a parameter set – often public key parameter sets used for RSA or Diffie-Hellman, and perform various tests upon each parameter to verify that it is correct. This could be a check that a prime parameter really is prime, or can include other checks such as ensuring that the subgroup generated by a generator is of the correct order.

However these primality tests are far from perfect. While there do exist deterministic primality tests that are able to correctly distinguish prime from composite with absolute certainty (even some proven to do so in polynomial time [4]) these algorithms are far too slow and impractical for everyday use, particularly on large inputs like those abundant in cryptography. We therefore rely upon probabilistic testing, which offers us great performance and practicality, but comes at a cost of accuracy. One vastly popular probabilistic primality test which sees use in nearly every cryptographic library and piece of mathematical software is the Miller-Rabin test [106, 135]. The accuracy of the Miller-Rabin test is very well understood, from Monier [108] and Rabin [135] we know that for an odd composite number n, a single round of the Miller-Rabin test will declare n as prime with probability no greater than $1/4$. Therefore by repeated testing we are able to reduce the error probability down to a very small margin of error.

A Miller-Rabin test requires as input two parameters, the number n we wish to test for primality, and an integer a with $1 \leq a < n$ which we call a *base*. For any odd $n > 1$, we can write $n = 2^e d + 1$ where d is odd. We know that if n is prime, then for any integer $1 \leq a < n$, either $a^d \equiv 1 \mod n$ or $a^{2^i d} \equiv -1 \bmod n$ for some $0 \leq i < e$. The Miller-Rabin test simply tests if either of these two conditions hold for the given base a. If neither condition holds, then we have proven n as composite otherwise, n is declared as *probably prime*. The Monier-Rabin [108, 135] theorem states that for any odd composite $n \neq 9$, the number of bases a for which either of the above two conditions hold is at most $\varphi(n)/4$ (where φ is Euler's totient function). A result of this is that the probability that an odd composite number n is declared prime by t rounds of the Miller-Rabin test is at most 4^{-t}. These are known as the *worst-case error estimates*, as they give the largest possible probability that a composite number is falsely classified as prime by the test.

In practice we may be presented with a large odd number n for which we are not sure

if it is prime or composite. Suppose that this number n has been chosen randomly from a set N_k of k-bit integers (as would be the case in random prime generation by primality testing). Say that we continue to randomly choose numbers n from N_k until we find one that passes t rounds (without failing any) of the Miller-Rabin test when choosing bases uniformly at random. From the Monier-Rabin theorem as given above, it may be tempting to think that the probability that this procedure returns a composite number is 4^{-t} for any bit-length k. However, as discussed in [17, 41], such a conclusion is fallacious. This is due to the distribution of prime numbers. In particular, because the primes are more sparsely distributed as k grows, it may be more likely to observe an event with probability 4^{-t} than the event that the randomly chosen number is prime. Moreover, the work of [41] goes on to show that it is a rare occurrence to find composite numbers n that meet the worst-case upper bound of $\varphi(n)/4$ on the number of bases that indicate n is prime. That is, for most n, the probability that a single round of Miller-Rabin would declare n as prime is considerably smaller than $1/4$. Therefore, when primality testing on random input of bit size k, we instead use bounds known as *average-case error estimates*. These take into consideration the distribution of the primes of this bit-size, and the rare occurrence of composites that reach the Monier-Rabin bound. The average-case error estimates are very useful in this context, as they can be used to give the number of rounds of Miller-Rabin that need to be performed on a random input to achieve a given error probability. We therefore see them used extensively when determining how many rounds of testing to perform when generating prime numbers in cryptographic libraries.

The distinction between working in the average-case or worst-case error estimates must however be very carefully considered. While the average-case estimates provide very precise error bounds when primality testing random input with Miller-Rabin, these bounds must be reduced back to the worst-case error estimates as soon as there is a possibility that input is not random. This distinction is perfectly illustrated by the two previously mentioned largest uses of primality testing in cryptographic applications: prime generation and prime checking. Many primality tests in cryptographic libraries are built for the main purpose of generating primes, and are therefore mostly testing input generated locally within the library. For example, *by default* the primality test in OpenSSL (pre-August 2018) utilises the average-case error estimates given in [41], which propagated into many informative sources such

as the Handbook of Applied Cryptography [105] and standards like the Digital Signature Standard (DSS) FIPS PUB 186-4 [84], to promise an error rate less than 2^{-80} when testing *random* input in OpenSSL's test. However, if the same primality test is made available as an external API or used elsewhere within the library, for example within a public key parameter checking function, the assumption that the numbers tested are random may no longer hold. In this case, we must work using instead the worst-case error estimates. These can drastically differ from the average-case. Again using the example of OpenSSL (pre-August 2018) the same primality test discussed above was used within their own Diffie-Hellman parameter checking function. Here OpenSSL called the test again using the average-case error estimates to calculate how many rounds of Miller-Rabin to perform, when the purpose of the test was to identify erroneous (or possibly malicious) parameters from an untrusted source. This meant that when testing a Diffie-Hellman prime parameter p for primality of size 1300 bits or more, just two rounds of Miller-Rabin were performed. Thus if a composite number that met Monier-Rabin bound was tested, it would have probability 1/16 of being accepted as prime by this parameter checking function. This is a vast difference from the assurance of error probability less than 2^{-80} given in the average-case setting. The difference in the error estimates can worsen even further if the implementation of the Miller-Rabin test chooses its bases for testing in a deterministic manner. In this case, carefully constructed malicious composite numbers can be *guaranteed* to be declared prime with any amount of repeated testing.

While it is the responsibility of the library to provide sufficient documentation to indicate how its primality test is being performed and give bounds on the error associated with probabilistic primality testing, often the onus of distinguishing between the use cases and error bounds ultimately falls upon the end-user of the library. In some cases, particularly to a user or developer who is not familiar with the subtle distinction between the theoretical bounds given, the documentation provided by many cryptographic libraries is not sufficient. Moreover, some of the APIs provided are insecure by default (like that seen in OpenSSL), or invite the user to misuse the test. Gutmann [70] identified the need to carefully define cryptographic APIs, recommending to "[p]rovide crypto functionality at the highest level possible in order to prevent users from injuring themselves and others through misuse of low-level crypto functions with properties they aren't aware of.". Later many others within the cryptographic community identified the need to design APIs which can be easily

used in a secure fashion (see [164, 68] as examples of this).

1.2 Contributions

In this thesis we study in depth primality tests and their use in cryptographic applications. We provide a systematic analysis of primality testing in a wide breadth of popular cryptography libraries and mathematical software. Our analysis ranges from the fundamental algorithms and the standards which give instruction for their use, to the implementation across cryptographic libraries and mathematical software, to the creation of carefully constructed composite numbers that have the highest probability of being declared prime by these tests. We present ways in which to create public key parameter sets that exploit the probabilistic nature of primality testing to produce malicious parameter sets for both elliptic curve and finite field Diffie-Hellman key exchange that offer an adversary an advantage in breaking the underlying hardness assumption of the scheme. It is the aim of this work to highlight attacks of this nature, with the aim of preventing them from being practical by promoting more resilient testing. We attempt to achieve this by outlining the key areas of error and misconception and using these to create new primality testing procedures that are efficient and secure across both the average-case and worst-case error settings. We also use our findings in primality testing as a case study for the design and implementation of a "misuse-resistant" API, that is, one which provides reliable results in all use cases even when the developer is crypto-naive.

1.2.1 Thesis Structure

In this section we highlight our main contributions, and give an overview of the chapters in this thesis and the problems by which they are motivated.

Chapter 2. This chapter provides the necessary preliminaries needed for the work to follow. We give an overview of primality testing, in which we introduce the four core primality tests we shall be discussing throughout the work. We also introduce a number of public-key cryptographic primitives and concepts that will be called on throughout the subsequent chapters. These are not designed to be comprehensive

introductions, but rather serve as a solid foundation for the rest of the thesis. We note that, where necessary, all other chapter-specific preliminaries are introduced in the relevant chapters.

Chapter 3. In this chapter we introduce known techniques for producing pseudoprimes for the Miller-Rabin and Lucas primality tests and extend them with our target applications in mind. We then survey the implementation landscape of primality testing in cryptographic libraries and mathematical software, evaluating their performance in the adversarial setting. We go on to examine the implications of our findings for applications, focussing on Diffie-Hellman parameter testing. The chapter ends with a discussion on avenues for improving the robustness of primality testing in the adversarial setting.

Chapter 4. In this chapter we extend our analysis on the generation of pseudoprimes for the Miller-Rabin test. We introduce methods to generate composite numbers with more than three prime factors, that are declared prime by a Miller-Rabin test with the highest probability. We then extend current methods of producing such numbers, to enable us to efficiently generate pseudoprimes with the particular properties required to form malicious Diffie-Hellman parameter sets that appear to be safe primes. These parameter sets pass standard approaches to parameter validation with some probability, but are designed such that the Discrete Logarithm Problem (DLP) is relatively easy to solve. We give an application of this malicious parameter set to OpenSSL (which will only accept safe prime parameters for DH) and password authenticated key exchange (PAKE) protocols. While the main focus of this chapter is on the finite field setting, we also briefly study malicious parameter sets based on pseudoprimes in the elliptic curve Diffie-Hellman (ECDH) setting. We show how such malicious ECDH parameters lead to attacks on PAKEs running over elliptic curves, as well as more traditional ECDH key exchanges.

Chapter 5. In this chapter we set out to design a performant primality test that provides strong security guarantees across all use cases and that has the simplest possible API. We examine different options for the core of our test, describing four different candidate primality tests and analysing them theoretically and experimentally. We then evaluate the performance of the chosen test in the use case of prime generation and discuss how our proposed test was fully adopted by the developers of

OpenSSL through a new API and primality test scheduled for release in OpenSSL 3.0 (2020).

Chapter 6. In this chapter we conclude and briefly mention avenues for further work.

1.2.2 OpenSSL Timeline

Throughout all three publications included in this thesis, we worked closely with the developers of OpenSSL. This included the disclosure of our findings before each publication, as well as an ongoing contribution to the development of OpenSSL, with the mitigations due to our results. As a consequence of this (and of course natural development), OpenSSL changed significantly over the three years of our analysis. This means that the parts of OpenSSL we analyse in one chapter may have changed significantly in the next.

Therefore in the process of consolidating this work into a thesis, we are sure to be as clear as possible on the version of OpenSSL being analysed at any one time. To this end, we now present a timeline of the different releases of OpenSSL relevant to this thesis, with a short explanation of the changes to OpenSSL made based on our contributions and their corresponding chapters.

Chapter 3. The work of Chapter 3 was mainly conducted in 2018. At this time OpenSSL were preparing the release of 1.1.1 - the new long term support (LTS) version of OpenSSL. Chapter 3 therefore analysed the most current version of the pre-release for 1.1.1 which was version 1.1.1-pre6 (May 2018) [122]. We note that the components studied were largely stable in other LTS releases such as 1.1.0h [120] and 1.0.2o [118] (to the extent in which the analysis performed still applies to the primality tests in these other versions), and remained similar to that of the early releases (version 0.9.5 of February 2000).

In August 2018 we reached out to OpenSSL to disclose the findings discussed in Chapter 3 before the work became public. At this time OpenSSL were in the process of amending their primality testing code to make it FIPS-complaint [117]. In light

of this, OpenSSL released version 1.1.1-pre9 [123], 1.1.0i [121] and 1.0.2p [119] (Aug 2018) that changed part of the primality testing code we had analysed (1.1.1pre7 and 1.1.1pre8 remained the same as 1.1.1pre6).

Because the changes made between 1.1.1-pre6 and 1.1.1-pre9, 1.1.0h and 1.1.0i, 1.0.2o and 1.0.2p were motivated by FIPS compliancy mostly made before our disclosure, they do not consider the adversarial scenario on which this chapter focuses, and therefore the default settings in OpenSSL remain weak in that scenario. While the documentation given for the affected APIs were updated at this time to reflect our findings, we will see how we further helped OpenSSL to address this issue in Chapter 5.

Chapter 4. The work of Chapter 4 was conducted simultaneously with the work done on Chapter 3 in 2018 but continued longer on into January 2019. Therefore this work again started analysis on OpenSSL 1.1.1-pre6, 1.1.0h and 1.0.2o, but also had to reflect the changes made to the primality tests in versions 1.1.1-pre9, 1.1.0i and 1.0.2p that occurred throughout this work. In Chapter 4 we therefore refer to versions of OpenSSL both before and after this update.

The update mainly concentrated on modifying the default rounds of Miller-Rabin performed by OpenSSL's primality test. More specifically, the function that gave the number of rounds of Miller-Rabin to perform based on the bit-size of the number being tested (see Table 3.4) was modified with the aim of achieving 128 bits of security instead of 80 bits (see Table 5.1). We note that previously to this, the last time these iteration counts were changed was in February 2000 (OpenSSL version 0.9.5), before which they were all 2, independent of the bit-size of the number being tested.

This work resulted in a contribution to the OpenSSL codebase by a pull request to increase the number of rounds of Miller-Rabin performed during the primality test on Diffie-Hellman parameters p and q during the check found in `DH_check`. This request was accepted by reviewers and merged into OpenSSL in March 2019 and was utilised as part of OpenSSL 1.1.1c [125] in May 2019.

Chapter 5. The work of Chapter 5 was conducted in 2019 after the release of OpenSSL version 1.1.1. We therefore studied the most recent version available at the time, which was OpenSSL 1.1.1c May 2019. The specific parts of the code studied remain almost completely unchanged in the most current version 1.1.1d [126] September 2019.

In June 2019 we contacted the OpenSSL developers to communicate the findings of Chapter 5. These suggestions were adopted with only minor modifications: the forthcoming OpenSSL 3.0 (scheduled for release in Q4 of 2020) will include our simplified API for primality testing, and the OpenSSL codebase has been updated to use it almost everywhere (the exception is prime generation, which uses the old API in order to avoid redundant trial division). Moreover, OpenSSL will now always use our suggested primality test (64 rounds of Miller-Rabin) on all inputs up to 2048 bits, and 128 rounds of Miller-Rabin on larger inputs. This represents the first major reform of the primality testing code in OpenSSL for more than 20 years.

Chapter 2

Background and Preliminaries

Contents

This chapter provides a mathematical background on various aspects of number theory and abstract algebra. We also give background material on the Diffie-Hellman and discrete log problems, elliptic curve cryptography, and on primality testing and its use in prime number generation

2.1 Mathematical Background

In this section we give a short introduction to the fundamental constructs in number theory and abstract algebra that are required by the following work. These are by no means a complete background, but aid in keeping the material required for the digestion of this thesis self-contained, and establish the chosen notation.

2.1.1 The Prime Number Theorem

The prime number theorem [143] gives an asymptotic form for the prime counting function $\pi(x)$, which counts the number of primes less than or equal to some integer

x. It states that:

$$\lim_{x\to\infty} \frac{\pi(x)}{x/\ln x} = 1.$$

This means that for large values of x, $\pi(x)$ is closely approximated by the expression $x/\ln x$. We shall be referring to this approximation often throughout this thesis.

2.1.2 The Group $\mathbb{Z}_n^*$

In modular arithmetic, the integers that are relatively prime to n in the set $\mathbb{Z}_n = \{0, 1, \ldots, n-1\}$ of n non-negative integers form a group under multiplication modulo n, called the *multiplicative group of integers modulo n*. This group is defined as, for any non-negative integer n

$$\mathbb{Z}_n^* = \{a \in \{0, 1, \ldots, n-1\} \mid \gcd(a, n) = 1\}.$$

This group is fundamental in number theory and has found applications in cryptography, integer factorisation, and primality testing. It is an abelian, finite group whose order is given by Euler's totient function $\varphi(n)$.

Definition 2.1 (Euler's totient function)**.** *Let n be a non-negative integer. Euler's totient function is defined as*

$$\varphi(n) = \#(\mathbb{Z}_n^*) = \#\{a \in \{0, 1, \ldots, n-1\} \mid \gcd(a, n) = 1\}.$$

In particular, if n is prime then $\mathbb{Z}_n^* = \{a \mid 1 \leq a \leq n-1\}$ and thus $\varphi(n) = n - 1$.

2.1.3 Cyclic Groups and Primitive Roots

We now introduce another important class of groups, known as cyclic groups. A *cyclic group* is a group that can be generated by a single element g (the group generator) and is formally defined as follows.

Definition 2.2 (Cyclic group)**.** *A group $\mathbb{G}$ is cyclic if there exists an element $a \in \mathbb{G}$ such that for each $b \in \mathbb{G}$ there is an integer i with $b = a^i$. Such an element a is called a generator of $\mathbb{G}$*

By Gauss [59] we know that the group $\mathbb{Z}_n^*$ is cyclic if and only if n is 1, 2, 4, p^k or $2p^k$, where p is an odd prime and $k > 0$. The generator of a cyclic group is known as *a primitive element of* $\mathbb{Z}_n^*$. In this thesis we are concerned only with groups of the form $\mathbb{Z}_p^*$ for some prime p, in this case a primitive element is more formally defined by the following theorem:

Theorem 2.1 (Primitive Root Theorem [145])**.** *Let p be a prime number. Then there exists an element $g \in \mathbb{Z}_p^*$ whose powers give every element of $\mathbb{Z}_p^*$, i.e.,*

$$\mathbb{Z}_p^* = \{1, g, g^2, \ldots, g^{p-2}\}.$$

Elements with this property are called primitive roots of $\mathbb{Z}_p$ or primitive elements of $\mathbb{Z}_p^*$.

2.1.4 Lagrange's Theorem

Over the next few sections we shall see how primitive elements are used within public-key cryptography through the Diffie-Hellman key exchange protocol. Aside from primitive elements, we also often require elements in the group that generate only a *proper* subgroup of size q (i.e. one in which the subgroup q is not equal to the whole group). To describe how to achieve this, we more formally introduce the notion of *order*.

Definition 2.3. *Let $\mathbb{G}$ be a group and $a \in \mathbb{G}$. The order of a is defined to be the smallest positive integer t such that $a^t = 1$, provided that such an integer exists. If such a t does not exist, then a is said to have infinite order.*

From the introduction of this definition we obtain a useful corollary of Theorem 2.1.

Corollary 2.1. Let p be a prime number. The primitive elements of $\mathbb{Z}_p^*$ are the elements of $\mathbb{Z}_p^*$ having order $p - 1$.

For a finite group $\mathbb{G}$, the order of a group $|\mathbb{G}|$ is the number of elements in $\mathbb{G}$, known as the cardinality of $\mathbb{G}$. We can now introduce the following.

Theorem 2.2 (Lagrange's Theorem [74])**.** *Let $\mathbb{G}$ be a finite group and let $a \in \mathbb{G}$. Then the order of a divides the order of $\mathbb{G}$. More precisely, let $n = |\mathbb{G}|$ be the order*

of $\mathbb{G}$ *and let* q *be the order of* a*, i.e.* a^q *is the smallest positive power of* a *that is equal to* 1*. Then*

$$a^n = 1 \quad \textit{and} \quad q \mid n.$$

By Lagrange's Theorem, we know that when working in the group $\mathbb{Z}_n^*$, the order of the group is $\varphi(n)$, and thus the order q of any element $a \in \mathbb{Z}_n^*$ must be such that $q \mid \varphi(n)$. If the order of the group is prime, say p, then we know that $\varphi(p) = p - 1$ and therefore the order q of any $a \in \mathbb{Z}_p^*$ must be such that $q \mid p - 1$.

2.1.5 Finding Primitive Elements and Elements of Order q

Finding primitive elements of $\mathbb{Z}_n^*$ or elements of order q with $q \mid \varphi(n)$ are both common requirements of cryptographic applications. Given a cyclic group $\mathbb{G}$ of order n, then for any divisor q of n we know that $\mathbb{G}$ has exactly $\varphi(q)$ elements of order q [58]. In particular, $\mathbb{G}$ has exactly $\varphi(n)$ generators, and hence the probability of a random element in $\mathbb{G}$ being a generator is $\varphi(n)/n$. Using the lower bound for the Euler totient function [140], this probability is at least $1/(6 \ln \ln n)$. We can therefore introduce Algorithm 1 as an efficient randomised algorithm for finding a generator of a cyclic group, given the prime factorisation of the group order n.

Algorithm 1 Finding a generator of a cyclic group

Input a cyclic group $\mathbb{G}$ of order n, and the prime factorisation $n = p_1^{e_1} p_2^{e_2} \cdots p_k^{e_k}$.
Output a generator g of $\mathbb{G}$.
Step 1: choose a random element g in $\mathbb{G}$.
1: $g \leftarrow_{\$} G$
Step 2: compute the order of g.
2: **for** $i = 1$ to k **do**
3: $\quad b \leftarrow g^{n/p_i}$
4: $\quad$ **if** $b = 1$ **then**
5: $\quad\quad$ go to Step 1
6: $\quad$ **end if**
7: **end for**
Step 3: output result.
8: Return g

If we instead wish to find an element of (high) order q, and not a generator then we may do the following: given a generator g in a cyclic group $\mathbb{G}$ of order n, and given a divisor q of n, an element h of order q can be efficiently obtained by computing

$h = g^{n/q}$. If q is a prime divisor of the order of n of a cyclic group $\mathbb{G}$, then we are able to find an element $h \in \mathbb{G}$ of order q without first having to find a generator of $\mathbb{G}$: select a random element $g \in \mathbb{G}$ and compute $h = g^{n/q}$; repeat until $h \neq 1$.

2.2 The Diffie-Hellman and Discrete Log Problems

The Diffie-Hellman (DH) problem sits at the very foundation of public-key cryptography [48]. Since its introduction, many cryptographic systems have been constructed from base assumptions on variants of the DH problem. Two of the most important variants are the computational Diffie-Hellman (CDH) and the decisional Diffie-Hellman (DDH) problems.

Definition 2.4 (CDH problem)**.** *Fix a cyclic group $\mathbb{G}$ and a generator $g \in \mathbb{G}$. Given elements $g^a, g^b \in \mathbb{G}$, the computational Diffie-Hellman problem is to compute an element h such that $h = g^{ab}$.*

Definition 2.5 (DDH problem)**.** *Fix a cyclic group $\mathbb{G}$ and a generator $g \in \mathbb{G}$. Given $g^a, g^b \in \mathbb{G}$ for uniformly and independently chosen a, b and a third element $g^z \in \mathbb{G}$, the decisional Diffie-Hellman problem is to decide if $g^z = g^{ab}$ or whether g^z was chosen uniformly at random from $\mathbb{G}$.*

The CDH assumption is a weaker assumption than the DDH assumption. This is due to the fact that if the CDH problem was easy, we would be able to compute g^{ab} from g^a and g^b and therefore given the tuple (g^a, g^b, g^z), distinguishing $g^z = g^{ab}$ from a uniform element in $\mathbb{G}$ would also be easy.

2.2.1 The Discrete Log Problem

Both the CDH and DDH problems are related to the discrete log problem (DLP).

Definition 2.6 (Discrete Log Problem (DLP))**.** *Given a group $\mathbb{G}$, a generator $g \in \mathbb{G}$ and an element $h \in \mathbb{G}$, the discrete log problem is to find an integer a such that $g^a = h$.*

If the discrete log problem in some group $\mathbb{G}$ is easy, then the CDH problem is too: given g^a and g^b, it would be possible to efficiently compute the discrete log $a = \log_g g^a$, then output g^{ab} by simply raising g^b to the power a. It therefore also follows that if the discrete log problem in some group $\mathbb{G}$ is easy, then the DDH problem is too. However, the converse of these statements does not seem to be true, and there are even examples of groups in which the discrete log and CDH problems are believed to be hard despite the DDH problem being easy (cf. [43, 101, 103]).

2.2.2 Prime Parameters and Safe Primes

Although there are various classes of cyclic groups in which the discrete log and Diffie-Hellman problems are believed to be hard, the most preferred are the cyclic groups of prime order. There are multiple reasons for this choice, with some based on usability features such as: finding a generator in prime order groups is trivial (whereas finding a generator of a non-prime order group requires the full factorisation of the order), and multiplicative inverses exist for all non-zero elements in a prime order group. But we also choose to use prime order groups for security benefit, as the discrete log problem is hardest in prime order groups. This is a consequence of the Pohlig-Hellman algorithm, which shows that solving the discrete log problem in a group of order q is easier if the group order is composed completely of small prime factors. More precisely, the cost of solving the discrete log problem using the Pohlig-Hellman algorithm is $O(q_i^{1/2})$ where q_i is the largest prime factor of the group order q.

To ensure that the discrete log problem is hard to solve, we can select a group order q that is prime, or contains a large prime factor. A typical recommendation is to work over $\mathbb{Z}_p^*$ where p is a safe prime, that is, to select $p = 2q+1$ for some prime q, where g should generate the group of order q modulo p. The size of the prime p is commonly chosen to be 1024, 2048, 3072 or 4096 bit. These sizes are chosen with respect to the industry's current understanding of the best known algorithms for solving the discrete log problem as well as an estimate of computational resources [3]. These primes chosen must be large enough to thwart subexponential algorithms for solving the discrete log problem such as the Number Field Sieve. For p that are not safe primes, the group order q can be much smaller than p. However, to maintain a high

level of security, q must still be large enough to thwart generic attacks, which for prime q run in time $O(q^{1/2})$. A common parameter choice is to use a 160-bit q with a 1024-bit p or a 224-bit q with a 2048-bit p. Diffie-Hellman parameters with p and q of these sizes are suggested for use and standardised in Digital Signature Algorithm (DSA) signatures in FIPS 186–4 [84].

We focus explicitly on how the security of schemes based on the Diffie-Hellman problem break down when we are able to trick implementations into believing they are working with prime parameters (both safe and non-safe) that are actually composite in Chapters 3 and 4.

2.2.3 Security Level and Generic Attacks

Throughout this thesis we shall often be referring to the security level, or the act of achieving n-bits of security. In general, a cryptographic system offers an n-bit security level if a successful *generic attack* can be expected to require an effort of approximately 2^n operations. A *generic attack* against a cryptographic primitive is one that can be run independently of the details of how that primitive is implemented. The security level is a measure for the security that may be attained and allows us to more explicitly define the difficulty of problems in terms of effort, and therefore elaborate on the terms "easy" or "hard". The security level is also particularly helpful when comparing different cryptographic schemes with each other, something we utilise when comparing finite field and elliptic curve Diffie-Hellman in Table 2.1 later in this section.

Generic attacks for solving the discrete log problem include algorithms such as Shanks' baby-step/giant-step method [144], Pollard's rho algorithm [131] and the Pohlig-Hellman algorithm [130].

2.2.4 Pollard Rho

The Pollard rho [132] method can be used to solve the discrete log problem $g^a = h$ in a finite group $\mathbb{G}$ where $g \in \mathbb{G}$ is a generator, $h \in \mathbb{G}$ and $a \in \mathbb{Z}$. We are interested

in the case where the group $\mathbb{G}$ is the specific group $\mathbb{Z}_p^*$ of nonzero residues modulo p for some prime $p > 3$. The Pollard rho method is based upon the observation that if one can find $a_j, b_j, a_k, b_k \in \mathbb{Z}_p^*$ such that $g^{b_j} h^{a_j} = g^{b_k} h^{a_k}$ with $a_j \neq a_k \pmod{p}$, then one can solve the discrete log problem as:

$$h = g^{(b_j - b_k)(a_k - a_j)^{-1} \pmod{p-1}} \tag{2.1}$$

To find such values we generate a pseudorandom sequence of integer pairs (a_i, b_i) modulo $(p-1)$ and a sequence of integers $x_i = h^{a_i} g^{b_i} \pmod{p}$, starting from the initial values $a_0 = b_0 = 0$, $x_0 = 1$ where the $i+1$ term is constructed from the ith term as:

$$(a_{i+1}, b_{i+1}) = \begin{cases} ((a_i + 1) \pmod{p-1}, b_i), & \text{if } 0 < x_i < \frac{1}{3}p, \\ (2a_i \pmod{p-1}, 2b_i \pmod{p-1}), & \text{if } \frac{1}{3}p < x_i < \frac{2}{3}p, \\ (a_i, (b_i + 1) \pmod{p-1}), & \text{if } \frac{2}{3}p < x_i < p, \end{cases}$$

and so

$$x_{i+1} = \begin{cases} hx_i \pmod{p}, & \text{if } 0 < x_i < \frac{1}{3}p, \\ x_i^2 \pmod{p}, & \text{if } \frac{1}{3}p < x_i < \frac{2}{3}p, \\ gx_i \pmod{p}, & \text{if } \frac{2}{3}p < x_i < p. \end{cases}$$

If we can find such a pair j, k with $j < k$ such that $x_j = x_k$, then we have $g^{b_j} h^{a_j} = g^{b_k} h^{a_k}$, and thus can use (2.1) to solve the discrete log problem for a. Floyd's cycle-finding algorithm [85] can be utilised to efficiently find such elements x_j, x_k with an expected running time $\mathcal{O}(\sqrt{p})$.

2.2.5 Pohlig-Hellman

We now give further details on the Pohlig-Hellman algorithm [130], due to its effectiveness in solving the discrete log problem in a group $\mathbb{G}$ when all non-trivial factors of the group order q are small, and known – something we shall utilise later in Chapter 4.

Suppose we are given a generator g of a group $\mathbb{G}$ of order q, an element $h \in \mathbb{G}$, and wish to find the discrete log x such that $g^x = h$. Given a factorisation of the group order $q = \prod_{i=1}^{k} q_i$, where q_i are pairwise relatively prime (this therefore does not need to be a complete factorisation), we know that

$$\left(g^{q/q_i}\right)^x = (g^x)^{q/q_i} = h^{q/q_i} \text{ for } i = 1, \ldots, k.$$

Therefore, setting $g_i = g^{q/q_i}$ and $h_i = h^{q/q_i}$, we have k instances of a discrete log problem $g_i^x = h_i$ in k smaller groups of order q_i respectively.

We then solve each of these k discrete log problems (using any suitable algorithm) to produce a set of solutions $\{x_1, x_2, \ldots, x_k\}$, with $x_i \in \mathbb{Z}_{q_i}$, for which $g_i^{x_i} = h_i = g_i^x$ and therefore $x = x_i \pmod{q_i}$ for all i.

The Chinese Remainder Theorem (CRT) allows us to uniquely determine the solution to the discrete log problem $x \pmod{q}$, using the k solutions $\{x_1, x_2, \ldots, x_k\}$, with $x_i \in \mathbb{Z}_{q_i}$, for which $g_i^{x_i} = h_i = g_i^x$.

Theorem 2.3 (Chinese Remainder Theorem)**.** *Given pairwise relatively prime integers $q_1, q_2 \ldots, q_k$ and arbitrary integers $x_1, x_2, \ldots, x_k$, the system of simultaneous congruences*

$$\begin{aligned} x &\equiv x_1 \pmod{q_1} \\ x &\equiv x_2 \pmod{q_2} \\ &\vdots \\ x &\equiv x_k \pmod{q_k} \end{aligned}$$

has a solution x. Moreover, the solution x is unique modulo $q = \prod_{i=1}^{k} q_i$ and can be found efficiently (in $O((\log n)^2)$ operations) by taking the least non-negative residue modulo q of

$$x = \sum_{i=1}^{k} x_i y_i z_i$$

where $y_i = q/q_i$ and the z_i are inverses defined by $z_i y_i \equiv 1 \pmod{q_i}$.

The implication of this is to choose to work in group q of prime order where possible.

2.2.6 Finite Field Diffie-Hellman Key Exchange

Diffie-Hellman key exchange provides a method of securely exchanging cryptographic keys over a public channel between two parties that require no prior knowledge of

each other. Suppose Alice and Bob want to participate in a key exchange. In finite-field Diffie-Hellman, Alice and Bob first agree on a prime p and a generator g of a multiplicative subgroup modulo p. Alice sends $g^a \pmod p$ to Bob and Bob sends $g^b \pmod p$ to Alice; each then computes a shared secret $g^{ab} \pmod p$. The CDH assumption states that an adversary observing the exchange cannot compute g^{ab}, furthermore the DDH assumption states that an adversary cannot distinguish g^{ab} from a random value g^z. However, Alice and Bob can easily compute $g^{ab} \pmod p$ by using their private exponents to compute $(g^b)^a \pmod p$ and $(g^a)^b \pmod p$ respectively.

2.3 Elliptic Curve Cryptography

In this section we introduce the basics of elliptic curves and elliptic curve cryptography (ECC). We are only concerned with a particular use case of elliptic curves for their use in Diffie-Hellman key exchange. We therefore only introduce the necessities required for further work in Chapter 4.

As well as being based on modular arithmetic and groups in finite fields, Diffie-Hellman can also be performed based upon groups consisting of points on an elliptic curve. While there exist sub-exponential algorithms for solving the discrete log problem over the multiplicative group of a finite field (such as Coppersmith's algorithm for $\mathbb{F}_{2^n}^*$ [36] or the number field sieve (NFS) [64, 79, 142]), there are currently no known[1] sub-exponential algorithms for solving the discrete log problem in appropriately chosen elliptic-curve groups. The consequence of this is that we are able to achieve n-bits of security using elliptic-curve groups of order $2n$-bits. This means that practically, elliptic-curve based Diffie-Hellman can use significantly smaller parameters than a finite field counterpart, resulting in more efficient implementations. See Table 2.1 for a direct comparison given by the US National Institute of Standards and Technology (NIST) [16].

[1] This is however a very active field of research, with recent breakthroughs including conjectured sub-exponential algorithms based upon heuristic assumptions and experimental evidence, see [128] and a broader discussion of the state of the art in [56].

Effective Key Length	Order-q Subgroup of $\mathbb{Z}_p^*$	Elliptic-Curve Group Order q
112	$p : 2048$, $q : 224$	224
128	$p : 3072$, $q : 256$	256
192	$p : 7680$, $q : 384$	384
256	$p : 15360$, $q : 512$	512

Table 2.1: A summary of the key lengths (in bits) recommended by NIST to achieve n-bits of security (effective key length) when using finite field and elliptic-curve discrete log based primitives.

2.3.1 Elliptic Curves

An elliptic curve over a prime field $\mathbb{F}_p$ (with $p > 3$) in short Weierstrass form is the set of solutions $(x, y) \in \mathbb{F}_p \times \mathbb{F}_p$ of an equation of the type $y^2 = x^3 + ax + b$, where $a, b \in \mathbb{F}_p$ satisfy $4a^3 + 27b^2 \neq 0$, together with the point at infinity $\mathcal{O}$.

By introducing a group operation (namely point addition) this set of solutions form an abelian group, called the *elliptic-curve group of E*. Let $P_1, P_2 \neq \mathcal{O}$ be points on a given curve E, with $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$. The addition rule is defined as follows [83].

1. If $x_1 \neq x_2$, then $P_1 + P_2 = (x_3, y_3)$ with

   $$x_3 = m^2 - x_1 - x_2 \pmod{p} \quad \text{and} \quad y_3 = m(x_1 - x_3) - y_1 \pmod{p},$$

   where $m = \frac{y_2 - y_1}{x_2 - x_1} \pmod{p}$.

2. If $x_1 = x_2$ but $y_1 \neq y_2$ then $P_1 = -P_2$ and so $P_1 + P_2 = \mathcal{O}$.

3. If $P_1 = P_2$ and $y_1 = 0$ then $P_1 + P_2 = 2P_1 = \mathcal{O}$.

4. If $P_1 = P_2$ and $y_1 \neq 0$ then $P_1 + P_2 = 2P_1 = (x_3, y_3)$ with

   $$x_3 = m^2 - 2x_1 \pmod{p} \text{ and } y_3 = m(x_1 - x_3) - y_1 \pmod{p},$$

   where $m = \frac{3x_1^2 + A}{2y_1} \pmod{p}$.

Consequently, for a non-negative integer k, it is possible to define the scalar multiplication $[k]P$ of a point P on the curve as the successive k-time addition of a point P with itself.

2.3.2 The Elliptic Curve Discrete Log Problem

Using the notation established above, we can now introduce the analogue of the discrete log problem for elliptic curves, known as the elliptic curve discrete log problem (ECDLP).

Definition 2.7 (Elliptic Curve Discrete Log Problem)**.** *Suppose E is an elliptic curve over $\mathbb{F}_p$ and $P \in \mathbb{F}_p \times \mathbb{F}_p$ is a point on E. Given a scalar multiple Q of P, the elliptic curve discrete log problem is to find $k \in \mathbb{Z}$ such that $Q = [k]P$.*

This problem forms the fundamental building block for elliptic curve cryptography, and has been a major area of research in computational number theory and cryptography for several decades. We now go on to describe one of its uses in the Diffie-Hellman key exchange protocol for elliptic curves.

2.3.3 Elliptic Curve Diffie-Hellman

We again consider the scenario in which two individuals, Alice and Bob, wish to securely create a shared cryptographic key over a public channel. For this we use elliptic curve Diffie-Hellman, described as follows.

Alice and Bob first agree on a public elliptic curve E and a public point $P \in E$ which generates a subgroup of order n. (In many scenarios, n is prime, or admits of a large prime factor. We look at precisely how the security of the scheme can be broken down when this is not the case, in Chaper 4.) Alice selects at random her private key $k_a \in [2, n-2]$ and computes her public key (a point on the elliptic curve E) $Q = [k_a]P$. She then sends this public key to Bob. Bob also selects at random his private key $k_b \in [2, n-2]$ and computes his public key $R = [k_b]P$. He sends his public key R to Alice. Alice and Bob can then construct their mutual shared key by computing $K = [k_a]R$ and $K = [k_b]Q$, respectively. Because of the group structure of the elliptic curve, we know these two K values are equivalent, since $[k_b]([k_a]P) = [k_b k_a]P = [k_a k_b]P = [k_a]([k_b]P)$.

2.4 Primality Testing

A primality test is an algorithm used to determine whether or not a given number is prime. Primality tests come in two different varieties: deterministic and probabilistic. Deterministic primality testing algorithms prove conclusively that a number is prime, but they tend to be slow and are not widely used in practice. A famous example is the AKS test [4]. We do not discuss such tests further in this thesis, except where they arise in certain mathematical software.

Probabilistic primality tests make use of arithmetic conditions that all primes must satisfy, and test these conditions for the number n of interest. If the condition *does not* hold, we learn that n must be composite. However, if it does hold we may only infer that n is *probably* prime, since some composite numbers may also pass the test. By making repeated tests, the probability that n is composite conditioned on it having passed some number t of tests can be made sufficiently small for cryptographic applications. A typical target probability is 2^{-80}, cf. [105, 4.49]. A critical consideration here is whether n was generated adversarially or not, since the bounds that can be inferred on probability may be radically different in the two cases; more on this in Chapters 3 and 4.

We now discuss four widely-used tests: the Fermat, Miller-Rabin, Lucas, and Baillie-PSW tests.

2.4.1 Fermat Test

The Fermat primality test is based upon the following theorem.

Theorem 2.4 (Fermat's Little Theorem)**.** *If p is prime and a is not divisible by p, then*

$$a^{p-1} \equiv 1 \pmod{p}.$$

To test n for primality, one simply chooses a base a and computes $a^{n-1} \pmod{n}$. If $a^{n-1} \not\equiv 1 \pmod{n}$, then we can be certain that n is composite. If after testing a variety of bases a_i, we find that that they all satisfy $a_i^{n-1} \equiv 1 \pmod{n}$, we may

conclude that n is probably prime.

It is well known that there exist composite numbers that satisfy $a^{n-1} \equiv 1 \pmod{n}$ for all integers a that are not divisible by n. These numbers completely thwart the Fermat test, and are known as Carmichael numbers. These will be of relevance in the sequel. The following result is fundamental in the construction of Carmichael numbers.

Theorem 2.5 (Korselt's Criterion [88])**.** *A positive composite integer n is a Carmichael number if and only if n is square-free, and $p-1 \mid n-1$ for all prime divisors p of n.*

2.4.2 Miller-Rabin Test

The Miller-Rabin [106, 135] (MR) primality test is based upon the fact that there are no non-trivial roots of unity modulo a prime. Let $n > 1$ be an odd integer to be tested and write $n = 2^e d + 1$ where d is odd. If n is prime, then for any integer a with $1 \leq a < n$, we have:

$$a^d \equiv 1 \mod n \quad \text{or} \quad a^{2^i d} \equiv -1 \bmod n \text{ for some } 0 \leq i < e.$$

The Miller-Rabin test then consists of checking the above conditions, declaring a number to be (probably) prime if one of the two conditions holds, and to be composite if both fail. If one condition holds, then we say n is a *pseudoprime to base a*, or that a is a *non-witness to the compositeness of n* (since n may be composite, but a does not demonstrate this fact).

For a composite n, let $S(n)$ denote the number of non-witnesses $a \in [1, n-1]$. By the following theorem, the exact number of non-witnesses $S(n)$ for any composite number n can be computed given the factorisation of n:

Theorem 2.6 (Monier [108], Proposition 1)**.** *Let n be an odd composite integer. Suppose that $n = 2^e \cdot d + 1$ where d is odd. Also suppose that n has prime factorisation $n = \prod_{i=1}^{m} p_i^{q_i}$ where each prime p_i can be expressed as $2^{e_i} \cdot d_i + 1$ with each d_i odd. Then:*

$$S(n) = \left(\frac{2^{\min(e_i) \cdot m} - 1}{2^m - 1} + 1 \right) \prod_{i=1}^{m} \gcd(d, d_i).$$

An upper-bound on $S(n)$ is given by results of [108, 135]:

Theorem 2.7 (Monier-Rabin Bound)**.** *Let $n \neq 9$ be odd and composite. Then*

$$S(n) \leq \frac{\varphi(n)}{4}$$

where φ denotes the Euler totient function.

This bound will be critical in determining the probability that an adversarially generated n passes the Miller-Rabin test. Since for large n, we have $\varphi(n) \approx n$, it indicates that no composite n can pass the Miller-Rabin test for t random bases with probability greater than $(1/4)^t$. The test is commonly implemented using either (a) a set of fixed bases (e.g. Apple corecrypto) or (b) randomly chosen bases (e.g. OpenSSL). Of course, the $(1/4)^t$ bound only holds in the case of randomly chosen bases.

2.4.3 Lucas Test

The Lucas primality test [15] makes use of Lucas sequences, defined as follows:

Definition 2.8 (Lucas sequence [9])**.** *Let P and Q be integers and $D = P^2 - 4Q$. Then the Lucas sequences (U_k) and (V_k) (with $k \geq 0$) are defined recursively by:*

$$\begin{aligned} U_{k+2} &= PU_{k+1} - QU_k \quad \textit{where,} & U_0 &= 0,\ U_1 = 1, \\ V_{k+2} &= PV_{k+1} - QV_k & V_0 &= 2,\ V_1 = P. \end{aligned}$$

The Lucas probable prime test then relies on the following theorem (in which $\left(\frac{x}{p}\right)$ denotes the Legendre symbol, with value 1 if x is a square modulo p and value -1 otherwise):

Theorem 2.8 ([39])**.** *Let P and Q be integers, $D = P^2 - 4Q$, and let the Lucas sequences $(U_k), (V_k)$ be defined as above. If p is a prime with $\gcd(p, 2QD) = 1$, then*

$$U_{p-\left(\frac{D}{p}\right)} \equiv 0 \pmod{p}. \tag{2.2}$$

The Lucas probable prime test repeatedly tests property (2.2) for different pairs (P, Q). This leads to the notion of a Lucas pseudoprime with respect to such a pair (here $\left(\frac{D}{n}\right)$ denotes the Jacobi symbol, since n is composite).

Definition 2.9 (Lucas pseudoprime)**.** *Let P and Q be integers and $D = P^2 - 4Q$. Let n be a composite number such that* $\gcd(n, 2QD) = 1$*. If $U_{n-\left(\frac{D}{n}\right)} \equiv 0 \pmod{n}$, then n is called a* Lucas pseudoprime *with respect to parameters (P, Q).*

We can now introduce the notion of a strong Lucas probable prime and strong Lucas pseudoprime with respect to parameters (P, Q) by the following theorem.

Theorem 2.9 ([9])**.** *Let P and Q be integers and $D = P^2 - 4Q$. Let p be a prime number not dividing $2QD$. Set $p - \left(\frac{D}{p}\right) = 2^k q$ with q odd. Then one of the following conditions is satisfied:*

$$p \mid U_q \quad or \quad \exists i \text{ such that } 0 \leq i < k \text{ and } p \mid V_{2^i q}. \tag{2.3}$$

The strong Lucas probable prime test repeatedly tests property (2.3) for different pairs (P, Q). This leads to the definition of a strong Lucas pseudoprime with respect to parameters (P, Q).

Definition 2.10 (strong Lucas pseudoprime)**.** *Let P and Q be integers and $D = P^2 - 4Q$. Let n be a composite number such that* $\gcd(n, 2QD) = 1$*. Set $n - \left(\frac{D}{n}\right) = 2^k q$ with q odd. Suppose that:*

$$n \mid U_q \quad or \quad \exists i \text{ such that } 0 \leq i < k \text{ and } n \mid V_{2^i q}.$$

Then n is called a strong Lucas pseudoprime *with respect to parameters (P, Q).*

A strong Lucas pseudoprime is also a Lucas pseudoprime (for the same (P, Q) pair), but the converse is not necessarily true. The strong version of the test is therefore seen as the more stringent and useful option.

Analogously to the Monier-Rabin theorem for pseudoprimes for the Miller-Rabin primality test, Arnault [9] gives a theorem on pseudoprimes to the strong Lucas test.

Theorem 2.10 (Arnault [9])**.** *Let D be an integer and n a composite number relatively prime to $2D$ and distinct from 9. For all integer D, the size*

$$SL(D, n) = \#\left\{(P, Q) \;\middle|\; \begin{array}{ll} 0 \leq P, Q < n, & P^2 - 4Q \equiv D \pmod{n}, \\ \gcd(Q, n) = 1, & n \text{ is } slpsp(P, Q). \end{array}\right\}$$

is less than or equal to $4n/15$ *except if* n *is the product*

$$n = (2^{k_1}q_1 - 1)(2^{k_1}q_1 + 1)$$

of twin primes with q_1 *odd and such that the Legendre symbols satisfy* $(D/2^{k_1}q_1 - 1) = -1$, $(D/2^{k_1}q_1 - 1) = 1$. *Here,* n *is an* $slpsp(P, Q)$ *denotes that* n *is a strong Lucas pseudoprime with respect to parameters* (P, Q). *Also, the following inequality is always true:*

$$SL(D, n) \leq n/2.$$

Therefore if (P, Q) are chosen at random from $0 \leq P, Q < n$ with $P^2 - 4Q \equiv D \pmod{n}$ and $\gcd(n, 2QD) = 1$, we can infer that t applications of the strong Lucas test would declare a composite n to be probably prime with a probability at most $(4/15)^t$ (with the exception of the specific twin primes mentioned in Theorem 2.10 which are declared prime with probability at most $(n/2)^t$ – however one can easily test for such an n).

2.4.4 Baillie-PSW

The Baillie-PSW test [134] is a probabilistic primality test consisting of a single Miller-Rabin test with base 2 followed by a single Lucas test. A slight variant of the test in which the Lucas test is replaced with a *strong* Lucas test is mentioned in [15]. Generally, the consensus that has emerged over time is that the Lucas test should be used with the parameters (P, Q) set as defined by Selfridge's method A:

Definition 2.11 (Selfridge's Method A [15])**.** *Let* D *be the first element of the sequence* $5, -7, 9, -11, 13, \ldots$ *for which* $\left(\frac{D}{n}\right) = -1$. *Then set* $P = 1$ *and* $Q = (1 - D)/4$.

If no such D can be found, then n must be a square and hence composite. In practice, one might attempt to find such a D up to some bound $D_{\max}$, then perform a test for squareness using Newton's method for square roots (see Appendix C.4 of [84]), before reverting to a search for a suitable D if needed. This is generally more efficient than doing a test of squareness first.

Since the parameters for both the Miller-Rabin and Lucas part of the test are fixed, the result of the Baillie-PSW test implemented this way is in-fact deterministic (in that the result of such a test will remain constant). We do however still classify Baillie-PSW as a probabilistic test.

The idea of this test is that the two components are "orthogonal" and so it is very unlikely that a number n will pass both parts. Indeed, there are no known composite n that pass the Baillie-PSW test. Gilchrist [60] confirmed that there are no Baillie-PSW pseudoprimes less than 2^{64}. PRIMO [99] is an elliptic curve based primality proving program that uses the Baillie-PSW test to check all intermediate probable primes. If any of these values were indeed composite, the final certification would necessarily have failed. Since this has never occurred during its use, PRIMO's author Martin estimates [158] that there are no Baillie-PSW pseudoprimes with less than about 10000 digits – yet this is just speculation.

This empirical evidence suggests that numbers of cryptographic size for use in Diffie-Hellman and RSA are unlikely to be Baillie-PSW pseudoprimes. However, Pomerance gives a heuristic argument in [133] that there are in fact infinitely many Baillie-PSW pseudoprimes. The construction of a single example is a significant open problem in number theory. There do not appear to exist any bounds demonstrating the test's strength on uniformly random k-bit inputs, in contrast to the results of [41] for the Miller-Rabin test. In summary, while the Baillie-PSW test appears to be very strong, there are no proven guarantees concerning its accuracy. One positive feature is that, being that we choose the parameters in a deterministic manner, it does not consume any randomness (whereas a properly implemented Miller-Rabin test does).

2.4.5 Supplementary and Preliminary Tests

It is often more efficient to perform some supplementary or preliminary testing on an input n before executing the main work of the primality test. A common strategy is to first perform trial division on n using a list of r small primes. This can be done directly, or by equivalently checking if $\gcd(\prod_i^r p_i, n) \neq 1$ where $\{p_1, \ldots, p_r\}$ is the list of primes used. The list of primes can be partitioned and multiple gcds computed,

so as to match the partial products of primes with the machine word-size. This is a very cheap test to perform and can be quite powerful when testing random inputs. The question arises of how r, the number of primes to use in trial division, should be set. We shall discuss this question in more detail in Chapter 5.

2.4.6 Standards and Technical Guidelines

Technical guideline documents such as Cryptographic Mechanisms BSI TR–02102–1 [55] and standards such as the Digital Signature Standard (DSS) FIPS 186–4 C.3.2 [84] and the International Standard for Prime Number Generation ISO/IEC 18032 [146] provide formal guidance and suggestions on primality testing algorithms and parameter choices.

BSI TR–02102–1 suggests that in the worst case, 50 rounds of random base selection Miller-Rabin must be performed, and in the average case it, like ISO/IEC 18032, references the method proposed by Damgård et al. [41] and the Handbook of Applied Cryptography [105] as described above. BSI TR–02102–1 also references the guidance given in FIPS 186–4, which suggests Miller-Rabin with more conservative numbers of rounds of iterations ($t = 40$ for 1024 and $t = 56$ for 2048 bit n) for DSA parameter generation, as well as giving a detailed justification. FIPS 186–4 advocates the use of an additional Lucas primality test (cf. Section 2.4.3) and also gives an elaboration of the distinction between the difference in the probability that a composite number survives t rounds of Miller-Rabin testing, with the probability that a number surviving t rounds of Miller-Rabin is composite. This is given explicitly as a warning when using error estimates in its Appendix F.2. While the current ISO/IEC 18032:2005 states correctly the worst case and average case error bounds, it does not make as clear the distinction between their use. This however seems to be addressed in the latest draft of ISO/IEC DIS 18032 (under development 2020), which gives a clear context for which each error condition can be applied in Annex A [147].

2.5 Prime Generation

A critical use case for primality testing is prime generation (e.g. for use in RSA keys). The exact details of the algorithms used vary across implementations, but the majority follow a simple technique based on first generating a random initial candidate n of the desired bit length k, possibly setting some of its bits, then doing trial division against a list of small primes, before performing multiple rounds of primality testing using a standard probabilistic primality test such as Miller-Rabin. If the trial division reveals a factor or the Miller-Rabin test fails, then another candidate is generated. This can be a fresh random value, but more commonly, implementations add 2 to the previous candidate n. This method is commonly known as an incremental search, and we give pseudocode of the algorithm in Algorithm 2.

Algorithm 2 Generating a random prime by incremental search

Input desired bit length k.
Output a k-bit prime.
Step 1: generate a random k-bit integer.
1: $n' \leftarrow \{0,1\}^{k-2}$
2: $n := 1||n'||1$ # this forces n to be odd and have exactly k bits
3: $n_{max} = \min(2^k, n + 2\mu)$
Step 2: test n for primality.
4: Run trial division and the Miller-Rabin test on n
5: **if** the output is *prime* **then**
6: Return n
7: **end if**
Step 3: increment n.
8: $n \leftarrow n + 2$
9: **if** $n > n_{max}$ **then**
10: go to Step 1
11: **else**
12: go to Step 2
13: **end if**

In order for this to be an efficient algorithm for generating primes, we must consider both the probability that a uniform k-bit integer is prime and how to efficiently test whether a given integer n is prime. The analysis in [31] provides choices for the length of the search such that error probability and failure probability are in practice negligible. We see these guidelines permeate through to standards, with ISO/IEC 18032:2005 standardising the incremental search algorithm, setting the choice for the size of the search as $\min(2^k, n + 2\mu)$ where $\mu = 10\ln(2^k)$ [146].

One might be concerned that an incremental search may not produce uniform prime output. For example, one may argue that primes p such that there exists another prime $p' < p$ with $p - p' < \delta$ for some small $\delta > 0$, are much less likely to be produced by incremental search. However, Brandt et. al. [31] strongly suggest that, compared to a uniform choice, there is no significant loss of security when using incremental search in cryptographic applications where secret primes are required.

2.5.1 Bias in Prime Generation

There are however prevalent examples of the use of biased algorithms for prime generation leading to cryptographic vulnerabilities. For example, a popular method for generating prime numbers (in this case for the eventual use in RSA public keys) is to construct a candidate of the form: $n = k \times M + 65537^a \pmod{M}$, where k, a are integers such that a is chosen randomly, k is a buffer used to ensure n is of the correct bit size and M is a primorial (a product of the first t successive primes), i.e. $M = \prod_{i=1}^{t} p_i$. This technique ensures that any candidate n produced would be such that $n \not\equiv 0 \pmod{p_i}$ for any of the t primes included in the primorial M, i.e. we achieve sieving over a list of small primes during the first stage. This structure is as described in [81, 80] and was widely deployed within the cryptographic library RSALib. This allowed an attack [114] in which knowledge of this specific structure allowed the factorisation of 1024 and 2048 bit RSA public keys in a widely used application.[2]

OpenSSL [124] provides a cryptographically secure strong pseudorandom number generator (see [149] for further analysis). However, aside from PRNG bugs like that seen in the Debian OpenSSL vulnerability [165], OpenSSL has also suffered from algorithmic biases that can be used to fingerprint primes as likely originating from OpenSSL. Mironov [107] observes that while OpenSSL's prime generation code contains methods both to generate safe primes (primes of the form $p = 2q + 1$ where q is also prime) and non-safe primes, an implementation bug caused part of the generation process for safe primes to be left over in the non-safe prime case. This meant that in the non-safe prime case, OpenSSL would output primes p such that

[2] Interestingly, we will be harnessing the techniques described here in Chapters 3 and 4 as we require both efficient and malleable methods to generate prime numbers when constructing pseudoprimes.

$p - 1 \not\equiv 1 \mod 3, 5, \ldots, 17863$. Although no vulnerably was found, this fingerprint was used by Mironov, and later Heninger et. al. [73], to attribute the source of factorisable RSA keys found in digital certificates to OpenSSL.

Prime generation, and the primality testing performed there within, will be discussed further in Chapter 5.

Chapter 3

Prime and Prejudice: Primality Testing Under Adversarial Conditions

Contents

In this chapter we provide a systematic analysis of primality testing under adversarial conditions, where the numbers being tested for primality are not generated randomly, but instead provided by a possibly malicious party. Such a situation can arise in secure messaging protocols where a server supplies Diffie-Hellman parameters to the peers, or in a secure communications protocol like TLS where a developer can insert such a number to be able to later passively spy on client-server data. We study a broad range of cryptographic libraries and assess their performance in this adversarial setting. As examples of our findings, we are able to construct 2048-bit composites that are declared prime with probability 1/16 by OpenSSL's primality testing in its default configuration; the advertised performance is 2^{-80}. We can also construct 1024-bit composites that *always* pass the primality testing routine in GNU GMP when configured with the recommended minimum number of rounds. And, for a number of libraries (Apple corecrypto and CommonCrypto, Cryptlib, LibTomCrypt, JavaScript Big Number, WolfSSL), we can construct composites that *always* pass the supplied primality tests. We explore the implications of these security failures in applications, focusing on the construction of malicious Diffie-Hellman pa-

rameters. We show that, unless careful primality testing is performed, an adversary can supply parameters (p, q, g) which on the surface look secure, but where the discrete logarithm problem in the subgroup of order q generated by g is easy. We close by making recommendations for users and developers. In particular, we promote increasing the number of rounds performed in the Miller-Rabin test to 64. This ensures that composite numbers are wrongly identified as being prime with probability at most 2^{-128}. We also suggest considering the use of the Baillie-PSW test if the additional code required is not too costly for the library.

3.1 Introduction and Motivation

Many cryptographic primitives rely on prime numbers, with RSA being the most famous example. However, even in constructions that do not rely on the difficulty of factoring integers into prime factors, primality is often relied upon to prevent an adversary from applying a divide-and-conquer approach (e.g. in the Pohlig-Hellman algorithm or in a Lim-Lee small subgroup attack [156]) or to prevent the existence of degenerate cases such as zero divisors (which may complicate security proofs or reduce output entropy).

One approach to obtaining prime numbers in instantiations of these cryptographic primitives is to produce such numbers as they are needed on whatever device requires them. This is accomplished by sampling random integers and checking for primality. This process can be computationally intensive to the point of being prohibitively so. The high cost of producing prime numbers led implementations to seek ways to reduce this cost and, as demonstrated in [114], these performance improvements may then lead to devastating attacks.

If the required prime numbers are public, an alternative approach is possible: (low-power) devices are provisioned with prime numbers from a server or a standard. For example, the popular Telegram messenger [93] uses Diffie-Hellman (DH) parameters provided *by the server* to establish end-to-end encryption *between peers*. If the peers do not validate the correctness of the supplied DH parameters,[1] the Telegram server can provide malicious DH parameters with composite group orders and

[1] We stress that they *do* perform validation in the default implementation.

thereby passively obtain the established secrets. As a concrete and closely related example, Bleichenbacher [26] showed how the reliance on a small and fixed set of bases in Miller-Rabin primality testing in GNU Crypto 1.1.0 could be exploited to fool GNU Crypto into accepting malicious DH parameters. In particular, this led to an attack on the GNU Crypto implementation of SRP enabling users' passwords to be recovered.

Another example is the Transport Layer Security protocol [47] which can use Diffie-Hellman key exchange to establish master secrets in the handshake protocol. The DH parameters are generated by the TLS server and sent to the client during each TLS handshake.[2] It is clear that the TLS server provider does not gain any advantage by sending malicious DH parameters to the client since it knows the established master key. However, we can consider an adversarial developer who implements a malicious server with backdoored DH parameter generation, cf. [161, 54]. If such parameters are accepted by TLS clients and used in the DH key exchange, a passive adversary can observe the traffic and obtain the master key. Here, weak DH parameters that still pass tests by trusted tools offer a sense of plausible deniability. Moreover, if an application simply silently rejects bad parameters then any countermeasures could be overcome by repeatedly sending malicious parameter sets having a reasonable probability of fooling those countermeasures, until the target client accepts them.

In recent years we have seen several backdoors in cryptographic implementations. For example, NIST standardised the Dual EC pseudorandom number generator (PRNG) which allows an adversary to predict generated random values if it can select a generator point Q whose discrete logarithm is known and collect enough PRNG output [34]. In 2016 it was shown that Juniper had implemented this PRNG in such a way as to enable an adversary to passively decrypt VPN sessions [33].

A notable example of a potential backdoor involving a composite number is the security advisory [137] pushed by command-line data transfer utility `socat`, which is popular with security professionals such as penetration testers. There, the DH prime p parameter was replaced with a new 2048 bit value because "*the hard-coded* 1024 *bit DH p parameter was not prime*". The advisory goes on to state "*since there is no indication of how these parameters were chosen, the existence of a trapdoor that*

[2] Up to version 1.2 (inclusive) of the protocol.

makes possible for an eavesdropper to recover the shared secret from a key exchange that uses them cannot be ruled out", which highlights a real world application of this attack model. Similarly, the prime group parameter p given by Group 23 of RFC5114 [90] for use in DH key exchanges has been found to be partially vulnerable to small subgroup attacks [156]. It might seem that code reviews and the availability of rigorous primality testing (in, say, mathematical software packages, cf. Section 3.4) would impose high rates of detectability for malicious parameter sets in code or standards, but as these examples highlight, such sets still occur in practice.

Given these incidents we can assume a motivated adversary who is able to implement software serving maliciously generated primes and/or DH parameters. Thus, there is a need for cryptographic applications that rely on third-party primes to perform primality testing. Indeed, many cryptographic libraries incorporate primality testing facilities and thus it appears this requirement is easy to satisfy. However, the primary application of these tests is to check primality (or, more precisely, compositeness) for locally-generated, random inputs during prime generation. Thus, it is a natural question to ask whether these libraries are robust against malicious inputs, i.e. inputs designed to fool the library into accepting a composite number as prime. We refer to this setting as *primality testing under adversarial conditions*.

3.1.1 Overview of Primality Testing

In this chapter we will be focused on the primality tests found in cryptographic libraries and mathematical software. The tests discussed are the Fermat, Miller-Rabin, Lucas and Baillie-PSW tests. While these tests were formally introduced in Chapter 2, we now discuss more specific details of their implementation within the libraries we study.

Clearly, when conducting a Miller-Rabin or Lucas test, the choice of the parameter t (the number of trials) is critical. Many cryptographic libraries, for example OpenSSL [124], use test parameters originating from [41] as popularised in the *Handbook of Applied Cryptography* [105]. These give the number of iterations of Miller-Rabin needed for an error rate less than 2^{-80}, when testing a *random* input n. A main result of [41] is that if n is a randomly selected b-bit odd integer, then t

independent rounds of Miller-Rabin testing to give an error probability:

$$P(X|Y_t) < b^{3/2}2^t t^{-1/2}4^{2-\sqrt{tb}} \quad \text{for } 3 \leq t \leq b/9 \text{ and } b \geq 21,$$

where X denotes the event that n is composite, and Y_t the event that t rounds of Miller-Rabin declares n to be prime. This bound enables the computation of the minimum value t needed to obtain $P(X|Y_t) \leq 2^{-80}$ for a range of bit-sizes b; see Table 3.4 later in this chapter.

However, these error estimates are for primality testing with Miller-Rabin on randomly generated n. In the *adversarial* setting, we are actually concerned with the probability that t trials of Miller-Rabin (or some other test) declare a *given* n to be prime, given that it is composite. This probability is independent of bit-size, and is at most $(1/4)^t$ if random bases are used in Miller-Rabin tests. Similar remarks apply for both variants of the Lucas test.

Many libraries, for example GNU GMP [66], provide primality testing functions to be deployed in applications such as mathematical software packages that require big integer arithmetic. These functions often obligate the user to choose the 'certainty' or accuracy of the primality test performed. Since these parameters are often hidden from the end user, this then forces the responsibility of choosing suitable parameters on the developer of the application using the library. The only resulting guidance that filters through from the standards is then found in the documentation of the library, which is often brief and informal.

3.1.2 Contributions & Outline

We investigate the implementation landscape of primality testing in both cryptographic libraries and mathematical software packages, and measure the security impact of the widespread failure of implementations to achieve robust primality testing in the adversarial setting.

In Section 3.2 we review known techniques for constructing pseudoprimes and extend them with our target applications in mind. In Section 3.3, we then survey primality testing in cryptographic libraries and mathematical software, evaluating their performance in the adversarial setting. We propose techniques to defeat their

tests where we can. Overall, our finding is that most libraries are not robust in the adversarial setting. Our main results in this direction are summarised in Table 3.3.

As one highlight of our results, we find that OpenSSL 1.1.1pre6 with its default primality testing routine will declare certain composites n of cryptographic size to be prime with probability 1/16, while the documented failure rate is 2^{-80}. This arises from OpenSSL's reliance on Table 3.4 to compute the number of rounds of Miller-Rabin testing required, and this number decreases as the size of n increases. As another highlight, we construct a 1024-bit composite that is guaranteed to be declared prime by the GNU GMP library [66] for anything up to and including 15 rounds of testing (the recommended minimum by GMP). This is as a result of GNU GMP initialising its PRNG to a static state and consequently using bases in its Miller-Rabin testing that depend only on n, the number being tested. We also show how base selection by randomly sampling from a fixed list of primes, as in Apple's corecrypto library, Cryptlib, LibTomCrypt, JavaScript Big Number (JSBN) and WolfSSL, can be subverted: we construct composites n of cryptographic size that are guaranteed to be declared prime by these libraries regardless of how many rounds of testing are performed.

We go on to examine the implications of our findings for applications, focussing on DH parameter testing. The good news is that OpenSSL is not impacted because of its insistence on safe primes for use in DH; that is, it requires DH parameters (p, q, g) for which $q = (p-1)/2$ and both p, q are tested for primality. Using the techniques in this chapter we cannot produce malicious parameters of this form. However, we will revisit the safe prime setting in Chapter 4, where the main focus will be to produce malicious safe prime DH parameters that are accepted by OpenSSL with some probability. In this chapter we show that when more liberal choices of parameter are permitted, as is the case in Bouncy Castle and Botan, we are able to construct malicious DH parameter sets which pass the libraries' testing but for which the discrete logarithm problem in the subgroup generated by g is easy.

We close by discussing avenues for improving the robustness of primality testing in the adversarial setting in Section 3.7.

3.2 Constructing Pseudoprimes

In this section, we review known methods of constructing pseudoprimes for the Miller-Rabin and Lucas tests. We also provide variations on these methods. We will use the results of this section in the next one, where we study the robustness of cryptographic libraries for primality testing in the adversarial setting.

3.2.1 Miller-Rabin Pseudoprimes

The exact number of non-witnesses $S(n)$ for any composite number n can be computed given the factorisation of n [108]. Generating composites n that have large numbers of non-witnesses is not so straightforward. In empirical work, Pomerance *et al.* [134] showed that many composite numbers that pass a Miller-Rabin primality test have the form $n = (k+1)(rk+1)$ where r is small and both $k+1$ and $rk+1$ are prime. More recently, Höglund [75] and Nicely [115] used the Miller-Rabin primality test as implemented in GNU GMP to test randomly generated numbers of this form for various values of r and for various different sizes of k. Their results support the claims made by [134].

We now consider existing methods for producing composites which have many non-witnesses, for two forms of the Miller-Rabin test: firstly where the bases are chosen randomly and secondly where a fixed set of bases is used.

3.2.1.1 Random Bases

For random bases, we are interested in constructing composite n that have large numbers of non-witnesses, i.e. for which $S(n)$ is large. Such numbers will pass the Miller-Rabin test with probability $S(n)/n$ per trial; of course, this probability is bounded by $\varphi(n)/4n \approx 1/4$ by the Monier-Rabin theorem, but we are interested in how close to this bound we can get.

Recall that Theorem 2.6 tells us for an odd composite integer n that has prime factorisation $n = \prod_{i=1}^{m} p_i^{q_i}$, where each prime p_i can be expressed as $2^{e_i} \cdot d_i + 1$ with

each d_i odd, we know that the number of non-witnesses $S(n)$ is:

$$S(n) = \left(\frac{2^{\min(e_i) \cdot m} - 1}{2^m - 1} + 1 \right) \prod_{i=1}^{m} \gcd(d, d_i)$$

where integers e, d are found by writing $n = 2^e \cdot d + 1$ where d is odd.

Note how the bound in this theorem does not depend on the exponents q_i, indicating that square-free numbers will have relatively large $S(n)$. Also note the dependence on the terms $\gcd(d, d_i)$, ensuring that the odd part of each prime factor p_i has a large gcd with the odd part of n is necessary to achieve a large value for $S(n)$. As an easy corollary of this theorem, we obtain:

Corollary 3.1 ([108])**.** Let x be an odd integer such that $2x+1$ and $4x+1$ are both prime. Then $n = (2x + 1)(4x + 1)$ has $\varphi(n) = 8x^2$ and achieves the Monier-Rabin bound, i.e. it satisfies $S(n) = \varphi(n)/4$.

The proof of this corollary follows easily on observing that we may take $m = 2$ and $d_1 = d_2 = x$ so that $\gcd(d, d_i) = x$ in the preceding theorem. Narayanan [113] extended work from Monier [108] who showed that if n is a Carmichael number of the form $p_1p_2p_3$, where each p_i is a distinct prime with $p_i \equiv 3 \pmod 4$, then $S(n)$ achieves the Monier-Rabin bound. Narayanan also gave further results showing that these two forms for n are the only ones achieving the Monier-Rabin bound, with all other n satisfying $S(n) \leq \varphi(n)/6$.

3.2.1.2 Fixed Bases

Some implementations of the Miller-Rabin primality test select bases from a fixed list (often of primes), rather than choosing them at random. For example, until an update motivated by the disclosure of this work, the primality test provided by Apple's CommonCrypto library `CCBigNumIsPrime` performed 16 rounds of Miller-Rabin using the first 16 primes as bases [6]. Similarly, Cryptlib, LibTomMath and WolfSSL all choose the first t entries from a hard-coded list of primes as bases when performing Miller-Rabin in their respective primality tests.

Arnault [8] presented a method for producing composite numbers $n = p_1 p_2 \cdots p_h$ that are guaranteed to be declared prime by Miller-Rabin for any fixed set of prime bases $A = \{a_1, a_2, \ldots, a_t\}$.

Not only is Arnault's method effective when an implementation chooses bases from a fixed list, it can also be utilised if an implementation chooses bases randomly from a large fixed list of possibilities. For example, an implementation might select prime bases randomly from a list of primes below 1000; since Arnault's method scales well (we simply need to solve more congruences simultaneously with the CRT) we can use this method to produce a composite n such that all primes below 1000 are non-witnesses for n. We shall see applications of this approach for different libraries in Section 3.3.

Since this approach is a very useful tool for us, we now go into more detail on the method proposed by Arnault, and give an example.

3.2.1.3 An Overview of Arnault's Method

Arnault's method generates n of the form $n = p_1 p_2 \ldots p_h$ where the p_i are distinct odd primes such that n is pseudoprime to a set of t prime bases $A = \{a_1, a_2, \ldots, a_t\}$. By [8, Lemma 3.2] we know that if $\gcd(a, n) = 1$ and $\left(\frac{a}{p_i}\right) = -1$ for all $1 \leq i \leq h$, then a will be a Miller-Rabin non-witness with respect to n (this set of conditions is sufficient but not necessary for a to be a Miller-Rabin non-witness with respect to n).

Now, by Gauss's law of quadratic reciprocity, we know that, for any prime p, $\left(\frac{a}{p}\right)$ can be determined from $\left(\frac{p}{a}\right)$ and the values of a and p taken modulo 4. This in turn means that, for each a, we can compute the set S_a of possible non-residues mod $4a$ of potential primes p. That is, we can compute the set S_a satisfying

$$\left(\frac{a}{p}\right) = -1 \iff p \bmod 4a \in S_a.$$

Arnault's method selects p_1 and then determines the other p_i from equations of the form $p_i = k_i(p_1 - 1) + 1$ where the k_i are values also chosen as part of the method (with $k_1 = 1$). This is done so as to ensure that the resulting $n = p_1 p_2 \ldots p_h$ is a

Carmichael number. But the conditions $\left(\frac{a}{p_i}\right) = -1$ for all $1 \leq i \leq h$ imply that, for each $a \in A$ and each $1 \leq i \leq h$ we have $k_i(p_1 - 1) + 1 \in S_a$. Rewriting this, we obtain that:

$$p_1 \bmod 4a \in \bigcap_{i=1}^{h} k_i^{-1}(S_a + k_i - 1), \tag{3.1}$$

where $k_i^{-1}(S_a + k_i - 1)$ denotes the set $\{k_i^{-1}(s + k_i - 1) \bmod 4a | s \in S_a\}$. This gives a set of conditions on the value of p_1 modulo $4a$ for each $a \in A$; typically a few candidates for $p_1 \bmod 4a$ remain for each value of a. By selecting one of these candidates z_a for each $a \in A$ and using the CRT, the conditions can be combined into a single condition on p_1 modulo $m = \text{lcm}(4, a_1, \ldots, a_t)$. The k_i values must be selected so that the sets on the right of (3.1) are non-empty; typically, they are set to small primes larger than the maximum of the $a \in A$ so that k_i^{-1} exists mod $4a$ for each a.

Arnault's method then brings into play other restrictions on $p_1 \bmod k_i$ for each $i = 2, \ldots, h$. These result from the requirement that n be a Carmichael number. We omit the full details, but, for example, when $h = 3$, the additional restrictions can be written as:

$$p_1 = k_3^{-1} \bmod k_2 \quad \text{and} \quad p_1 = k_2^{-1} \bmod k_3$$

Making the k_i co-prime to each other and to the $a \in A$ ensures that another application of the CRT can be made to incorporate these conditions. The end result is a single condition of the form:

$$p_1 = z \bmod \text{lcm}(4, a_1, \ldots, a_t, k_2, \ldots, k_h)$$

where z is a fixed value determined by the choice of the z_a values and the additional restrictions.

Finally, the method repeatedly generates candidates for p_1 satisfying the above constraint and uses the equations $p_i = k_i(p_1 - 1) + 1$ to determine the other p_i. The method is successful for a given p_1 if all of the resulting $p_1, \ldots, p_h$ are prime.

Evidently, the method is complex and not guaranteed to succeed on every attempt for a given set A. However, it can be iterated with different choices of the k_i until the sets on the right of (3.1) are non-empty; moreover a back-tracking approach can be used

Table 3.1: Values a and subsets S_a of residues modulo $4a$ of primes p such that $\left(\frac{a}{p}\right) = -1$.

a	S_b
2	$\{3, 5\}$
3	$\{5, 7\}$
5	$\{3, 7, 13, 17\}$
7	$\{5, 11, 13, 15, 17, 23\}$
11	$\{3, 13, 15, 17, 21, 23, 27, 29, 31, 41\}$
13	$\{5, 7, 11, 15, 19, 21, 31, 33, 37, 41, 45, 47\}$
17	$\{3, 5, 7, 11, 23, 27, 29, 31, 37, 39, 41, 45, 57, 61, 63, 65\}$
19	$\{7, 11, 13, 21, 23, 29, 33, 35, 37, 39, 41, 43, 47, 53, 55, 63, 65, 69\}$
23	$\{3, 5, 17, 21, 27, 31, 33, 35, 37, 39, 45, 47, 53, 55, 57, 59, 61, 65, 71, 75, 87, 89\}$
29	$\{3, 11, 15, 17, 19, 21, 27, 31, 37, 39, 41, 43, 47, 55, 69, 73, 75, 77, 79, 85, 89, 95, 97, 99, 101, 105, 113\}$

to select the z_a values to speed-up the entire process of constructing p_1. The density of all-prime solutions $(p_1, \ldots, p_h)$ amongst all possible candidates $(p_1, \ldots, p_h)$ satisfying $p_1 = z \bmod \text{lcm}(4, a_1, \ldots, a_t, k_2, \ldots, k_h)$ and $p_i = k_i(p_1 - 1) + 1$ for $i = 2, \ldots, h$ can be estimated using standard heuristics concerning the distribution of primes of size $L = \text{lcm}(4, a_1, \ldots, a_t, k_2, \ldots, k_h)$; it is roughly $1/(\log^h(L) \cdot \sum_{i=2}^{h} \log(k_i))$.

Notice that, the larger the set A, the larger the modulus L in the condition determining p_1 will be. Thus, if A contains many bases, then larger p_i and hence larger n will tend to result. Moreover, all-prime solutions will become less dense. As an example, when analysing the primality test in Maple V.2, Arnault [8] considers $h = 3$ so $n = p_1p_2p_3$ and $A = \{2, 3, 5, 7, 11\}$ (so $t = 5$); he works with $k_2 = 13$ and $k_3 = 41$ and arrives finally at the condition:

$$p_1 = 827443 \bmod 4924920.$$

For $p_1 = 286472803$, this yields a 29-decimal digit composite passing Maple's fixed-base Miller-Rabin primality test.

We give a short example of the method described for an n of the form $n = p_1p_2p_3$ for which the first 10 primes are Miller-Rabin non-witnesses. That is, we target $A = \{2, 3, 5, 7, 11, 13, 17, 19, 23, 29\}$.

We start by generating the set S_a of residues modulo $4a$ of primes p such that $\left(\frac{a}{p}\right) = -1$ for each base $a \in A$.

Table 3.1 gives the sets S_a for our chosen set A.

a	$\bigcap_{i=1}^{h} k_i^{-1}(S_a + k_i - 1)$	Modulo
2	$\{\mathbf{3}, 5\}$	8
3	$\{\mathbf{7}\}$	12
5	$\{\mathbf{3}, 7, 13, 17\}$	20
7	$\{\mathbf{15}\}$	28
11	$\{21, \mathbf{23}\}$	44
13	$\{21, \mathbf{47}\}$	52
17	$\{5, 29, \mathbf{31}, 39, 63, 65\}$	68
19	$\{33, 37, 39, \mathbf{47}, 69\}$	76
23	$\{31, \mathbf{47}, 57, 87, 89\}$	92
29	$\{19, 37, 41, \mathbf{55}, 77, 95, 99, 113\}$	116

Table 3.2: Values a and the sets $\bigcap_{i=1}^{h} k_i^{-1}(S_a + k_i - 1)$ when $k_2 = 41$ and $k_3 = 101$.

We now set $k_2 = 41$ and $k_3 = 101$; these are coprime to all $a \in A$. We find subsets of the S_a that meet the requirement:

$$p_1 \pmod{4a} \in \bigcap_{i=1}^{h} k_i^{-1}(S_a + k_i - 1).$$

This gives us a set of residues modulo $4a$ for each $a \in A$ that p_1 must satisfy. We give an example of this for the first 10 primes in Table 3.2.

We then need to make a choice of one residue z_a per set. This choice is arbitrary, but we note that not all combinations of choices will lead to a solution. We give an example of a good set of choices in Table 3.2 in bold.

We then have two additional conditions to add, based on our choice of the k_i values. These can be written as:

$$p_1 = k_3^{-1} \bmod k_2 \quad \text{and} \quad p_1 = k_2^{-1} \bmod k_3$$

In our example, we chose $k_1 = 41$ and $k_2 = 101$ which gives us:

$$p_1 \equiv 28 \pmod{41} \qquad \text{and} \qquad p_1 \equiv 32 \pmod{101}.$$

We can then use the Chinese Remainder Theorem to simultaneously solve for the 10 conditions implied by the bold entries in Table 3.2 and the two conditions above. In this case, we have the solution:

$$p_1 \equiv 36253030834483 \bmod 107163998661720.$$

The prime

$$p_1 = 1424453871614154824048263654181759622666891330061 63$$

satisfies this condition, and yields primes

$$p_2 = 584026087361803477859788098214521445293425445325 2643$$
$$p_3 = 143869841033029637228874629072357721889356024336 22363$$

such that the product $n = p_1 p_2 p_3$ is a 512-bit number that is a Miller-Rabin pseudoprime to the bases $2, 3, 5, 7, 11, 13, 17, 19, 23$ and 29.

3.2.1.4 Hybrid Technique

The method above produces composites that are in fact always Carmichael numbers. We know from Section 3.2.1.1 that if n is a Carmichael number with 3 distinct prime factors all congruent to 3 (mod 4), then n has the maximum number of non-witnesses, $\varphi(n)/4$. We can set $h = 3$ in Arnault's method and tweak it slightly to ensure that, as well as producing n with a specified set A of non-witnesses, it produces an n meeting the Monier-Rabin bound, so that random base Miller-Rabin tests will also pass with the maximum probability. The tweak is very simple: we ensure that $2 \in A$; this forces $p_1 \equiv 3$ or 5 (mod 8); we then select $p_1 \equiv 3$ (mod 8) so that $p_1 \equiv 3$ (mod 4). Arnault's method sets $p_i = k_i(p_1 - 1) + 1$ where the k_i are co-prime to all the elements of A. Since $2 \in A$, the k_i must all be odd; it is easy to see that this forces $p_i \equiv 3$ (mod 4) too.

We will give an application of this technique in Section 3.3.11.

3.2.1.5 Extension For Composite Fixed Bases

The method of Arnault [8] works (as presented) only for prime bases, and not for *composite* bases. Although less common, some implementations use both prime and composite bases in their Miller-Rabin testing. By setting $n \equiv 3$ (mod 4), we know that $e = 1$ when writing $n = 2^e \cdot d + 1$ for d odd. In this case, the conditions to pass the Miller-Rabin test simply become $a^{(n-1)/2} \equiv \pm 1 \pmod{n}$. Hence, if $n \equiv 3$ (mod 4)

is pseudoprime to some set of bases $\{a_1, a_2, \ldots a_t\}$, then n is also pseudoprime for any base b arising as a product $b = a_1^{e_1} \cdot a_2^{e_2} \cdot \cdots \cdot a_t^{e_t} \pmod{n}$ (for any set of indices $e_i \in \mathbb{Z}$). Therefore we can construct a composite n that is pseudoprime with respect to any list of bases $\{b_1, \ldots, b_t\}$ (of which any number can be composite) by using the hybrid method described in Section 3.2.1.4, but with set A in that method being the complete set of prime factors arising in the b_i. Note that in this method, n is of the form $n = p_1 p_2 p_3$ where each $p_i \equiv 3 \pmod 4$, so we have $n \equiv 3 \pmod 4$ as needed. Moreover, because of the form of n, the composites generated in this manner will also meet the Monier-Rabin bound.

We will give an application of this technique in Section 3.3.3, where we study Mini-GMP [66] which uses Euler's polynomial to generate Miller-Rabin bases.

3.2.2 Lucas Pseudoprimes

Like Miller-Rabin pseudoprimes, Lucas pseudoprimes are with respect to some choice of test parameters. Throughout this work we follow Selfridge's Method A [15] of parameter selection, which is summarised as follows:

Definition 3.1 (Selfridge's Method A [15])**.** *Let D be the first element of the sequence $5, -7, 9, -11, 13, \ldots$ for which $\left(\frac{D}{n}\right) = -1$. Then set $P = 1$ and $Q = (1-D)/4$.*

There are two reasons for studying this particular method for setting parameters. The first is that it is the parameter choice used when performing the Lucas part of the Baillie-PSW primality test [134, 15]. The second is that this is the method that both Java [38] and Crypto++ [40] libraries that we study use in their implementation of the Lucas test.

The Lucas and strong Lucas-probable prime tests with this parameter choice are commonly referred to in the literature as Lucas and strong Lucas-Selfridge probable prime tests. Pseudoprimes for this parameter choice are well-documented. The OEIS sequence A217120 [13] presents a small list of them, referring to a table of all Lucas pseudoprimes below $10^{14} \approx 2^{47}$ compiled by Jacobsen [76]. There is an equivalent sequence A217255 [14] for strong Lucas pseudoprimes. Any pseudoprime

for the strong Lucas probable prime test with respect to some parameter set (P, Q), is also a pseudoprime for the Lucas probable prime test.

Arnault [8] also presented a scalable method that takes as input a set of parameter choices $\{(P_1, Q_1, D_1), (P_2, Q_2, D_2), \ldots, (P_t, Q_t, D_t)\}$ and returns a composite n of the form $n = p_1 p_2 \cdots p_h$ that is a strong Lucas pseudoprime to the parameters (P_i, Q_i, D_i) for all $1 \leq i \leq t$. The method is similar to that for constructing Miller-Rabin pseudoprimes for fixed bases, but differs in its details. In particular, the two construction methods are sufficiently different that it seems hard to derive a single method producing n that are pseudoprimes for both the Miller-Rabin and Lucas tests.

3.2.2.1 A Specialisation of Arnault [8] for Selfridge's Method A

For Selfridge's Method A, we know that if we take an n such that $\left(\frac{5}{n}\right) = -1$, then a single test on n with parameter set $(P, Q, D) = (1, -1, 5)$ will be performed. We next show how to specialise Arnault's construction [8] so that it will produce composites n that are guaranteed to be declared prime by a strong Lucas test for this parameter set.

Following Arnault's construction, we consider n of the form $n = p_1 p_2 p_3$ where $p_i = k_i(p_1 + 1) - 1$ for $i \in \{2, 3\}$, with k_2 and k_3 odd integers.

We first note that the p_i must satisfy certain conditions with respect to Legendre symbols (see [8, Lemmas 6.1 and 6.2]):

$$\left(\frac{D}{p_i}\right) = \left(\frac{Q}{p_i}\right) = -1 \qquad \text{for all } i \text{ such that } 1 \leq i \leq 3.$$

With our single parameter set $(P, Q, D) = (1, -1, 5)$, this becomes:

$$\left(\frac{-1}{p_i}\right) = \left(\frac{5}{p_i}\right) = -1 \qquad \text{for all } i \text{ such that } 1 \leq i \leq 3. \tag{3.2}$$

Now $\left(\frac{-1}{p_i}\right) = -1 \Leftrightarrow p_i \equiv 3 \pmod 4$. Since $p_i = k_i(p_1 + 1) - 1$ for $i \in \{2, 3\}$, and the k_i are odd, then it is easy to show that if $p_1 \equiv 3 \pmod 4$ then it follows that $p_i \equiv 3 \pmod 4$ for $i = 2, 3$ as well. We also have that $\left(\frac{5}{p_i}\right) = -1 \Leftrightarrow p_i \equiv 2 \text{ or } 3 \pmod 5$.

Therefore condition (3.2) is satisfied when $p_1 \equiv 3$ or $7 \pmod{20}$ (by the CRT) and $p_i \equiv 2$ or $3 \pmod 5$ for $i \geq 2$.

At this point we must choose k_2, k_3 and add conditions that ensure the coefficients in [8, Lemma 6.1] are indeed integers. These conditions are simple:

$$p_1 \equiv k_3^{-1} \pmod{k_2} \qquad \text{and} \qquad p_1 \equiv k_2^{-1} \pmod{k_3}.$$

We choose to fix $p_1 \equiv 7 \pmod{20}$ and select $(k_2, k_3) = (31, 43)$.

This produces our final congruence that prime p_1 must satisfy: $p_1 \equiv 6647 \pmod{26660}$. We now search for a prime p_1 that satisfies this congruence, and such that p_2 and p_3 satisfying $p_i = k_i(p_1 + 1) - 1$ for $i = 2, 3$ are also primes with $p_2 \equiv p_3 \equiv 2$ or $3 \pmod 5$.

The smallest solution is the following:

$$p_1 = 486527, p_2 = 15082367, p_3 = 20920703$$

This yields a 68-bit $n = 153515674455111174527$ which indeed does pass the strong Lucas test using Selfridge's Method A for parameter selection. Of course, we can take any (p_1, p_2, p_3) satisfying the above conditions (which are not too onerous to satisfy), and in this sense the method scales well to numbers n of cryptographically interesting size.

This generation technique is also versatile, as we can simply include additional parameters in our set dependent on which parameter selection methods a particular test uses. This allows us to generate composites that are declared prime by a variety of strong Lucas tests, at the small cost of solving a few more simultaneous congruences with the CRT.

A Large Strong Lucas Pseudoprime. To show just how well this method scales to produce numbers of a cryptographically interesting size, we use our SAGE implementation of the method as described above to construct an n of the form $n = p_1 p_2 p_3$,

where $p_i = k_i(p_1 + 1) - 1$ with $(k_2, k_3) = (31, 43)$ and

$$\begin{aligned} p_1 = 2^{576} &\cdot \texttt{0x000000000000000000000000822d00000000000000000000000000} \\ + 2^{384} &\cdot \texttt{0x000000001a0900000000000000000000000000000000000000} \\ + 2^{0} &\cdot \texttt{0x00000000000000000000000000000000000000000000007254e7cb}. \end{aligned}$$

Then $n = p_1 p_2 p_3$ is a 2048-bit strong Lucas pseudoprime for Selfridge's Method A of parameter selection.

3.3 Cryptographic Libraries

Many cryptographic libraries offering implementations of common cryptographic protocols also provide a toolkit for handling arbitrary-precision integer arithmetic, including primality testing. These functions would be used, for example, for testing the primality of Diffie-Hellman parameters.

This section provides a survey of primality testing in a broad and representative range of cryptographic libraries (OpenSSL, GNU GMP and Mini-GMP, NSS, Apple corecrypto and CommonCrypto, Cryptlib, JavaScript Big Number (JSBN), LibTomMath, LibTomCrypt, WolfSSL, Libgcrypt, Java, Bouncy Castle, Botan, Crypto++ and Golang). For each library, we first describe how it implements primality testing. We then tailor a composite likely to be declared prime by each particular library, and quantify the probability that our composite passes the library's primality test (so that the primality test fails). Our findings are summarised in Table 3.3. Throughout this chapter, we will refer to the number of rounds of Miller-Rabin testing as t.

3.3.1 OpenSSL

OpenSSL is the most widely used open source cryptographic library and TLS implementation. Throughout, we consider OpenSSL 1.1.1-pre6 [122] (May 2018) as this is the most current pre-release of the next long term support (LTS) version OpenSSL 1.1.1. We note that the components studied are largely stable to other LTS releases such as 1.1.0h [120] and 1.0.2o [118] (to the extent in which the analysis performed

Table 3.3: Results of our analysis of cryptographic libraries. This shows how the number of rounds of Miller-Rabin used is determined, whether a Baillie-PSW test is implemented, the documented failure rate of the primality test (that is, the probability that it wrongly declares a composite to be prime), and our highest achieved failure rate for composite input.

Library	Rounds of MR	BPSW?	Documented Failure Rate	Our Highest Failure Rate
Apple CommonCrypto	16	No	$< 2^{-32}$	100%
Apple corecrypto	User-defined $t \leq 256$	No	$(1/4)^t$	100%
Botan 2.6.0	User-defined t	No	$\leq (1/2)^t$	$(1/4)^t$
Bouncy Castle C# 1.8.2	User-defined t	No	$(1/4)^t$	$(1/4)^t$
Cryptlib 3.4.4	User-defined $t \leq 100$	No	Not given	100%
Crypto++ 7.0	2 or 12	Yes	Not given	0%
GNU GMP 6.1.2	User-defined t	No	$(1/4)^t$	100% for $t \leq 15$
GNU Mini-GMP 6.1.2	User-defined t	No	$(1/4)^t$	100% for $t \leq 101$
Golang 1.10.3	User-defined t	Yes	$< (1/4)^t$	0%
Golang pre-1.8	User-defined t	No	$< (1/4)^t$	100% for $t \leq 13$
Java 10	User-defined t	Yes[‡]	$< (1/2)^t$	0% for ≥ 100 bits
JSBN 1.4	User-defined t	No	$< (1/2)^t$	100%
Libgcrypt 1.8.2	User-defined t	No	Not given	1/1024[†]
LibTomCrypt 1.18.1	User-defined $t \leq 256$	No	$(1/4)^t$	100%
LibTomMath 1.0.1	User-defined $t \leq 256$	No	$(1/4)^t$	100%
NSS 3.50	User-defined t	No	Not given	100% for $t \leq 10$[††]
OpenSSL 1.1.1-pre6	Default bit-size based	No	$< 2^{-80}$	1/16
WolfSSL 3.13.0	User-defined $t \leq 256$	No	$(1/4)^t$	100%

[†] When calling the `check_prime` function as opposed to `gcry_prime_check` (or calling `gcry_prime_check` in versions prior to 1.3.0).
[‡] When testing input of size at least 100 bits.
[††] Results of testing do not appear to be consistent across different machines.

still applies to the primality tests in these other versions), and remain similar to that of the early releases (version 0.9.5 of February 2000).

Analysis. The primality tests in OpenSSL reside in the crypto library, which also houses a wide range of implementations of cryptographic algorithms. The services provided by the crypto library are used by the OpenSSL implementations of SSL, TLS and S/MIME, and have also been used to implement SSH, OpenPGP, and other cryptographic standards.

The functions called upon to perform primality testing in the OpenSSL BIGNUM library are `BN_is_prime_ex` and `BN_is_prime_fasttest_ex` found in `bn_prime.c`. The bulk of the primality testing algorithm is done in `BN_is_prime_fasttest_ex` where

Table 3.4: The rounds t of Miller-Rabin performed chosen by OpenSSL when testing b-bit integers with `checks` = `BN_prime_checks`.

b	t	b	t
$b \geq 1300$	2	$400 > b \geq 350$	8
$1300 > b \geq 850$	3	$350 > b \geq 300$	9
$850 > b \geq 650$	4	$300 > b \geq 250$	12
$650 > b \geq 550$	5	$250 > b \geq 200$	15
$550 > b \geq 450$	6	$200 > b \geq 150$	18
$450 > b \geq 400$	7	$150 > b$	27

$t =$`checks` rounds of Miller-Rabin are performed, each with a randomly chosen base. The `checks` variable is provided as a parameter to the primality verification function. The function `BN_is_prime_ex` simply calls `BN_is_prime_fasttest_ex` without doing any trial divisions. The composites n that we produce have factors much larger than those in the trial divisions that OpenSSL performs. This means that, for our purposes, the result of calling either function is equivalent. Therefore we will focus only on `BN_is_prime_fasttest_ex`.

Number of Miller-Rabin rounds. Both primality testing functions allow the user to determine the rounds of Miller-Rabin performed. The documentation indicates that if the user sets the value of `checks` to the variable `BN_prime_checks`, then the number of Miller-Rabin iterations t is chosen such that the probability of a Miller-Rabin test declaring a *random* composite number n as prime is less than 2^{-80}. The number of rounds performed is then based on the bit-size b of the number n being tested. The relationship between these two values is shown in Table 3.4. The entries here are based on average case error estimates taken from the Handbook of Applied Cryptography [105], which in turn references [41].

Base Selection. OpenSSL chooses the Miller-Rabin bases it uses in a pseudorandom manner, by using OpenSSL's function `BN_rand_range()` with an optional flag set to `PRIVATE`. This then calls `bnrand` to generate a pseudorandom base a in the range $1 \leq a < n$ using a cryptographically strong pseudorandom number generator with entropy inputs gathered from the operating system, cf. [149] for details on OpenSSL's random number generation.

Pseudoprimes. As mentioned in Section 3.1, the average case estimates from [41] are designed only to be used on testing numbers during prime generation. Indeed, OpenSSL correctly applies primality testing as outlined above in this situation. However, we found nothing in the documentation to warn about the adversarial setting. Instead it appears to be left up to the user to decide how many rounds of testing are needed, and if they set `checks = BN_prime_checks` then Table 3.4 would dictate how many rounds are applied.

In this setting, we are able to undermine OpenSSL's guarantees by producing composite numbers using the methods described in Section 3.2.1.1. That is, we can easily construct numbers of the form $n = (2x + 1)(4x + 1)$ with x odd and $2x + 1, 4x + 1$ prime, and be sure that n will pass random-base Miller-Rabin tests with probability roughly $1/4$ per test. For example, for n having $b = 2048$ bits, OpenSSL will apply $t = 2$ tests, and we have a $1/16$ chance of our composite n deceiving OpenSSL.

An Example Pseudoprime for OpenSSL.
Let

$$\begin{aligned} x = 2^{960} &\cdot \texttt{0x000000000000000000000000000000058971a1a5c1b26e7} \\ + 2^{768} &\cdot \texttt{0x26a401f6937c462b9c3460e000000000000000000000000} \\ + 2^{192} &\cdot \texttt{0x000000000000000000000000000000000000000000058971a} \\ + 2^{0} &\cdot \texttt{0x1a5c1c624a1531999dbd1b6f5bbcb1942e8c4f8138f86231}. \end{aligned}$$

Then $n = (2x + 1)(4x + 1)$ produces a 2048-bit composite number that is declared prime by OpenSSL's `BN_is_prime_fasttest_ex` with `checks = BN_prime_checks` with probability $1/16$.

3.3.2 GNU GMP

The GNU Multiple Precision Arithmetic Library [66], GNU GMP or simply GMP, is a popular open source arbitrary precision integer library that is widely deployed in mathematical software packages. We consider the latest version GMP 6.1.2 throughout.

Analysis. GMP provides its own datatype to handle big integers known as `mpz_t`. GMP's primality test is implemented in `mpz_probab_prime_p(mpz_t n, int reps)`. On input n, this function performs some trial divisions, then a fixed-base Fermat test with base $210 = 2 \cdot 3 \cdot 5 \cdot 7$, and finally $t =$ `reps` rounds of Miller-Rabin; the latter is implemented in function `mpz_millerrabin`. The value of `reps` is selected by the caller. The documentation gives assurance that a composite number will be identified as being prime with a probability of less than $(1/4)^{\texttt{reps}}$ and states that "reasonable values of `reps` are between 15 and 50".

Base Selection. GMP uses a pseudorandom number generator (PRNG) to choose the base used for each Miller-Rabin test. The PRNG's state is initialised in the function `mpz_millerrabin` by calling `gmp_randinit_default(rstate)`, which uses the Mersenne Twister algorithm. This initial seed state is then used as a source of randomness in `mpz_urandomm(a, rstate, n)` to generate a uniform random integer base a between 2 and $n - 2$ inclusive.

While GMP offers to seed PRNGs and to explicitly pass the state to functions requiring access to pseudorandom numbers, this option is not available for primality testing, i.e. each call to `mpz_millerrabin` will work with an identical PRNG state. Thus, since the initial seed state is constant, the resulting sequence of a values chosen by `mpz_urandomm` for a fixed n is also constant. Note, though, that different a may be chosen for different n, since the bases a are sampled uniformly in a range depending on n. This, in effect, means that the bases chosen when testing n are defined as a function of n. Therefore the result of `mpz_probab_prime_p(mpz_t n, int reps)` for fixed values of n and t is deterministic.[3]

Pseudoprimes. For integers n, t, let $(a_1, a_2, \ldots, a_t)$ denote the deterministic list of bases used by GMP, where $t =$ `reps`. By setting $n = (2x+1)(4x+1)$ with x odd and $2x+1, 4x+1$ both prime, we will obtain a number for which random base MR tests will pass with probability roughly $1/4$. Since $(a_1, a_2, \ldots, a_t)$ is pseudorandom, we may expect that an n constructed in this way would pass the MR tests in GMP with probability $(1/4)^t$. Thus, for example, for the minimum recommended value of

[3] We note that the same sequence of a_i may still be produced even for different n when n is only slightly smaller than a power of two. This is due to the application of rejection sampling by comparison with n to sample in a range up to n.

$t = 15$, it might be feasible to construct a suitable n which would always be declared prime by just trying sufficiently many random values of x.

However, recall that we need $2x + 1$ and $4x + 1$ to be simultaneously prime, and we must also pass the base 210 Fermat test. This makes the cost of constructing n prohibitively high with this direct approach, since the probability that random x will give prime pairs $(2x + 1, 4x + 1)$ is approximately $(2/\ln x)^2$, and the special form of n means that a Fermat test will pass with probability roughly $1/2$, while passing t rounds of MR testing will happen with probability only $(1/4)^t$. Putting this together, each x would pass with probability about $1/2^{2t-1}(\ln x)^2$; for a 99% chance of success in finding a good x with $\ln x = s$, we would need about $5 \cdot 2^{2t-1}s^2$ trials, each trial involving at least a primality test on $2x + 1$. For a 1024-bit n and $t = 15$ trials (the minimum recommended by GMP), roughly 2^{47} trials would be needed, each involving at least a 512-bit primality test.

Instead, and partly inspired by the ROCA attack [114] and the form of the primes exploited there [81, 80], we consider x of the special form $x = kM + 189$ where M is a product of the first ℓ primes in the set $\mathcal{P} = \{2, 3, \ldots, 373\}$ and k is a randomly chosen integer of a size to make $n = (2x + 1)(4x + 1)$ have the desired target size (say, 1024 bits). Here ℓ is a parameter to be chosen later. The selection of x of this form ensures that $2x + 1 = 2kM + 379$ and $4x + 1 = 4kM + 757$ are *not* divisible by the first ℓ primes in $\mathcal{P}$, boosting the chances that $2x + 1$ and $4x + 1$ are both prime (the form of x essentially ensures that $2x + 1, 4x + 1$ pass trial divisions for the first ℓ primes in $\mathcal{P}$; here we rely on the fact that 379 and 757 are both prime and larger than 373). The offset of 189 is specially chosen so that the Fermat test on n to base 210 will always pass for n of the chosen form. This follows from a bespoke mathematical analysis that we now discuss, before giving an example pseudoprime for GMP.

Constructing GMP Pseudoprimes. Recall that we work with candidates x of the form $x = kM + 189$, and then consider $n = (2x + 1)(4x + 1)$; we select x so that $2x + 1$ and $4x + 1$ are both prime, and we select M as a product of the first ℓ primes from the set $\mathcal{P} = \{2, 3, \ldots, 373\}$. We justify this construction here.

First, note that $2x + 1 = 2kM + 379$ while $4x + 1 = 4kM + 757$, where both 379 and 757 are prime. Considering $2x + 1$ modulo each of the ℓ prime factors p in M, we see that $2x + 1 = 379 \bmod p \neq 0 \bmod p$ because $p < 379$; similarly, we obtain $4x + 1 = 757 \bmod p \neq 0 \bmod p$. Hence no such p divides either $2x + 1$ or $4x + 1$, so these numbers are not divisible by any of the primes in the product M (i.e. the first ℓ primes). For this reason, with random choices of k and with $x = kM + 189$, it follows that $2x + 1$ and $4x + 1$ are more likely to be prime than they would be for random choices of x. An analysis of the effect involves an application of the inclusion-exclusion principle to determine how many numbers are "sieved out" by the process. We omit the full analysis here, but note that, for numbers of cryptographically interesting size and with $\ell = 69$ that we use in the construction of our 1024-bit example for n, the effect is to increase the probability of primality for each number from $2/\ln x$ to roughly $10/\ln x$. Since we have two numbers $2x + 1$, $4x + 1$ whose primality behaves largely independently over the choice of x, this yields a 25-fold improvement in the performance of our approach over the direct approach of trying random x values. This speed up is discussed more extensively in Section 4.3.4 of Chapter 4 and can be calculated from Equation 4.4.

Next, we consider the Fermat test on n with base $a = 210$, assuming the factors $2x + 1$ and $4x + 1$ are prime. This test computes the value of $a^{n-1} \bmod n$ and compares it to 1. Now $n - 1 = (2x + 1)(4x + 1) = 8x^2 + 6x = 2x(4x + 3)$, so we obtain:

$$a^{n-1} = (a^{4x+3})^{2x} = 1 \bmod 2x + 1$$

and

$$a^{n-1} = a^{8x^2+6x} = (a^{2x+1})^{4x} \cdot a^{2x} = 1 \cdot a^{2x} = a^{2x} \bmod 4x + 1.$$

Here, we have made repeated use of Fermat's Little Theorem (which states that $a^{p-1} = 1 \bmod p$ for prime p and $a \neq 0 \bmod p$).

It follows that $a^{n-1} = 1 \bmod n$ if and only if a is a quadratic residue modulo $4x+1$. Hence n passes a Fermat test to base a for roughly half of the possible bases a (since roughly half of the values $a \bmod n$ are quadratic residues mod$4x + 1$).

Now we use the fact that $a = 210 = 2 \cdot 3 \cdot 5 \cdot 7$ to write:

$$\left(\frac{210}{4x+1}\right) = \left(\frac{2}{4x+1}\right)\left(\frac{3}{4x+1}\right)\left(\frac{5}{4x+1}\right)\left(\frac{7}{4x+1}\right).$$

Since M is even, we can write $4x{+}1 = 8k(M/2){+}757 = 5 \bmod 8$, hence $(\frac{2}{4x+1}) = -1$. Also $(\frac{3}{4x+1}) = (\frac{4kM+757}{3}) = (\frac{757}{3}) = (\frac{1}{3}) = 1$, where we use Gauss's Law of Quadratic Reciprocity and $3|M$. Similarly, we obtain $(\frac{5}{4x+1}) = -1$ and $(\frac{7}{4x+1}) = 1$. Combining everything, we finally get

$$\left(\frac{210}{4x+1}\right) = (-1) \cdot 1 \cdot (-1) \cdot 1 = 1.$$

We conclude that the Fermat test for n of the given form with base $a = 210$ always passes.

A Pseudoprime for GMP. Our code for constructing x (and n) of this special form first picks a target bit-size for n, then selects ℓ as large as possible so that there are enough choices for k for there to be sufficiently many candidates that one suitable x will result. For each resulting x, our code tests $2x+1$ and then $4x+1$ for primality, and (if these tests pass) applies the GMP primality test for the desired number of t rounds of MR testing.

For n of 1024 bits, we set $\ell = 69$, taking M as the product of the primes up to 349, and leaving a 51-bit value for k. The choice of M increases the probability that both of $2x+1$ and $4x+1$ are prime by a factor of roughly 25, and the form of x ensures that the Fermat test always passes, giving another factor of 2 improvement. Using a total of 33,885 core-hours (3.87 core-years) of computation in parallel on 872 cores running at 2.4GHz (kindly donated by CloudFlare), we found the following 1024-bit example passing GMP's primality test with $t = 15$ rounds of MR testing:

$$\begin{aligned} n = 2^{960} &\cdot \texttt{0x000000000000000000000000000081d564fbdd20b406} \\ + 2^{768} &\cdot \texttt{0x750af7bd334dcf547b131a1d8f8235fd603dba44e22e0775} \\ + 2^{576} &\cdot \texttt{0x0ecf755051d33cb8895413f5d69f5a3df701889e3a69f92e} \\ + 2^{384} &\cdot \texttt{0xdd3f5f36662521877231ba4753a3e7185a89ddb0b2d73a35} \\ + 2^{192} &\cdot \texttt{0x9e976a9bcfeae1a7c026d74bc7515a5010f3cd62c69fa9ad} \\ + 2^{0} &\cdot \texttt{0x7b699f40e7a85192e1a4aa95537363fcb93d789aee32bbbf}. \end{aligned}$$

We recall that this n will *always* pass GMP's primality testing for 15 MR rounds because the generation of the MR bases depends deterministically on n.

3.3.3 Mini-GMP

Mini-GMP is a small implementation of a subset of GMP's `mpn` and `mpz` interfaces included within GMP 6.1.2 [66]. This library includes its own miniature implementation of `mpz_probab_prime_p(n, reps)`. The most significant change compared to GMP is that Miller-Rabin testing is performed explicitly with a deterministic sequence of t bases obtained by evaluating Euler's polynomial $a(x) = x^2 + x + 41$ at $x = 0, 1, 2, \ldots, t-1$. It also omits GMP's Fermat test.

Pseudoprimes. The use of a sequence of deterministic bases in Mini-GMP enables us to predict the bases that will be chosen for any particular value t of `reps`. The bases are not all prime (though Euler's polynomial famously does produce many primes), so we cannot directly use Arnault's method from Section 3.2.1.2. Instead, we use our extension for composite, fixed bases method in Section 3.2.1.5.

Using this approach, we constructed a 2960-bit composite $n = p_1 p_2 p_3$ that passes up to $t = 101$ rounds of Mini-GMP's Miller-Rabin testing. Of the 101 bases produced by Euler's polynomial, 86 were already primes and the remaining 15 bases all factorised into various combinations of the four primes $163, 167, 179$ and 199. The combined list of 90 unique primes was then used with the method described in Section 3.2.1.5 to produce n. This n is given in the example below.

We note that the documentation for Mini-GMP is shared with the main GMP library, implying to a user that 15 to 50 rounds of MR testing would be reasonable.

A Pseudoprime for Mini-GMP. Using our SAGE implementation of the composite fixed base technique as described in Section 3.2.1.5, we construct an n of the

form $n = p_1p_2p_3$, where $p_i = k_i(p_1 - 1) + 1$ with $(k_2, k_3) = (10937, 11257)$ and

$$\begin{aligned} p_1 = 2^{960} &\cdot \texttt{0x000000000000000000000000000000000000000000000002e394} \\ + 2^{768} &\cdot \texttt{0x1a2fe4aa9e66358347f63732494d08635ccc9ae0a3c17764} \\ + 2^{576} &\cdot \texttt{0xa8e266f4d26758ab804a702c235f63b1e109a81fc007f94b} \\ + 2^{384} &\cdot \texttt{0xec5158f231a30b1cbf96a7fc444c09be62f5a809f049cc5d} \\ + 2^{192} &\cdot \texttt{0xe94b84275c38885c9b61a6bdc44111501527722a8ac87ea2} \\ + 2^{0} &\cdot \texttt{0xa5d4498caa2d9d07b34001a508fa53063991206268c547d7}. \end{aligned}$$

This yields a 2960-bit composite n that is guaranteed to pass any number up to and including $t = 101$ rounds of Mini-GMP's primality test. This large example was created to emphasise how the recommended number of rounds of MR to perform by GMP's documentation can be vastly surpassed.

However, we are also able to generate numbers of a more suitable bit-size for cryptographic use. Using the same SAGE implementation we construct an n of the form $n = p_1p_2p_3$, where $p_i = k_i(p_1 - 1) + 1$ with $(k_2, k_3) = (10709, 10781)$ and

$$\begin{aligned} p_1 = 2^{576} &\cdot \texttt{0x000000000000000000000031f56286a48b9e806b0a4dc84} \\ + 2^{384} &\cdot \texttt{0xbbf808788471fb4b91c291e92f25617f832581dd28b88325} \\ + 2^{192} &\cdot \texttt{0xd8d391bc68e6b720ef5a6f6701d8845658af13436b63217f} \\ + 2^{0} &\cdot \texttt{0x71d60fade1aaea8eaf28b3c2ac81b9233d18fc962a7761b3}. \end{aligned}$$

This produces the following 2048-bit composite n that is guaranteed to pass any number up to and including $t = 70$ rounds of Mini-GMP's primality test. This is

still well above the advised 15-50 rounds.

$$
\begin{aligned}
n = 2^{1920} &\cdot \texttt{0x0000000000000000d17cafbd9fb7f539a99b9f970e13e0e8} \\
+ 2^{1728} &\cdot \texttt{0xac245b6a8937b4047d70858d6aea422f2d00f84a52906bb6} \\
+ 2^{1536} &\cdot \texttt{0xf39431237ae9f21a341337d37cf88b8668a91313c3bbb5e1} \\
+ 2^{1344} &\cdot \texttt{0xe3243ba1d22a22b81b497befd0dbb83bfe88e269438ceb8c} \\
+ 2^{1152} &\cdot \texttt{0x22fe8211696dae60f6770064904c5675accd31933f686727} \\
+ 2^{960} &\cdot \texttt{0xe325348a2d42394d50708924257b2a38141e035333ae12ab} \\
+ 2^{768} &\cdot \texttt{0xe9161b0757c5fec92dab42c507126d0ed02cab6cd69879c6} \\
+ 2^{576} &\cdot \texttt{0x5cd68172318fe5a28805a24a3a71a7135b9582cafd5ea8e1} \\
+ 2^{384} &\cdot \texttt{0x6a733b98bf41d770b80fc5067d80c44b86b73707764f176b} \\
+ 2^{192} &\cdot \texttt{0xab00cb70531918ca06fe9c8096dd98db1bace156d0222e1a} \\
+ 2^{0} &\cdot \texttt{0x28217bf86d18a4c08c28f579c566d2adbe1bf5721d2334cb}.
\end{aligned}
$$

3.3.4 NSS

Mozilla's Network Security Services (NSS) [109] is a set of libraries designed to support cross-platform development of security-enabled client and server applications. NSS is the TLS library used by Mozilla's Firefox browser. Applications built with NSS can also support SSL v3, various PKCS, X.509 v3 certificates and many other security standards. We consider NSS 3.50 (Feb. 2020) throughout.

Analysis. NSS provides the primality test `mpp_pprime` found in `mpprime.c`. This function takes as input a number n to be tested and the number of rounds of testing chosen by the user, t. Each round of testing is composed of a Miller-Rabin test performed on a base a, where a is chosen by the function `mpp_random`. The function `mpp_random` uses the C library's `rand()` function to generate random values. In the corresponding documentation for `mpp_random`, it is noted that "[i]t is up to the caller to seed this generator before it is called.", yet there is no seeding of this generator before its use in `mpprime.c`. Therefore, this pseudorandom number generator will output a deterministic sequence of numbers, and thus the Miller-Rabin test will be performed on the same bases each time `mpp_pprime` is invoked (in a manner similar

to that found in GMP above).

Pseudoprimes. As was the case with GMP: if a composite number n is found such that the first x bases produced by the function `mpp_random` when testing n are non-witnesses, then n will be declared prime for $t \leq x$. Due to the similarities in the GMP method of base selection, we are able to construct pseudoprimes for `mpp_pprime` by modifying the code used for the generation of the examples in Section 3.3.2. To do this, we simply change the call to the primality test in GMP with `mpp_pprime`.

However, unlike deterministic sequence of bases seen in GMP, the return values of `rand()` do not appear to be consistent across different machines. This reduces the portability of pseudoprimes to `mpp_pprime`, as the result of performing the test may be specific to just one machine. Furthermore, depending on the platform and configuration, the underlying arbitrary precision integer arithmetic library used by NSS (known as `mpi`) uses different limbs [110], which may change the number of `rand()` calls (and thus its output). While this may affect the usability of these pseudoprimes, it is still conceivable that a particular pseudoprime could be constructed with the sole purpose of being accepted by just one machine, that could for example be a particularly desirable host.

An Example Pseudoprime for NSS. By modifying the C implementation described to efficiently generate pseudoprimes to GMP (described in Section 3.3.2) to instead make calls to `mpp_pprime` in NSS, we were able to produce an example pseudoprime for NSS. Using approximately 60 core-hours on 3.2GHz CPUs we produced a 1024-bit composite number n of the form $n = (2x + 1)(4x + 1)$ that passed the primality test provided by NSS with $t = 10$ round of MR testing, with

$$\begin{aligned} x = 2^{384} &\cdot \texttt{0x000000000000000004071079147a638c3701eed9a97d0267e} \\ + 2^{192} &\cdot \texttt{0x6a7a7744256f79b5fb0f420fb2623219a17775639f052cdf} \\ + 2^{0} &\cdot \texttt{0xff70763848269b02017b92484b65779743b2f6bcbfcaed3d}. \end{aligned}$$

However, due to the inconsistency of results across machines, this was declared prime by NSS with $t = 10$ rounds of MR testing *only* on the single machine with which it

was produced (our particular target machine). No further attempts were made to produce an example that was declared prime across multiple machines.

3.3.5 Apple corecrypto and CommonCrypto

Apple's CommonCrypto [6] library provides iOS and OS X cryptographic services. CommonCrypto relies upon Apple's corecrypto library [7] to provide implementations of low-level cryptographic primitives. The Apple corecrypto does not have explicit version numbers, but we are able to give the various versions of Apple's operating systems which were using the corecrypto library analysed here in August 2018. Therefore, the results here affect versions prior to iOS 12.1, macOS Mojave 10.14.1, tvOS 12.1, watchOS 5.1, iTunes 12.9.1, and iCloud for Windows 7.8.

Analysis. Apple's primality test can be found in corecrypto file `ccz_is_prime.c`, which contains the function `ccz_is_prime`. This function takes as input a number n to be tested and the number of rounds of testing chosen by the user, t. This function then calls the function `ccprime_rabin_miller` found in `ccprime_rabin_miller.c`. This in turn optionally checks that the number being tested is odd, is not one of the first 256 primes, and not divisible by one of the first 119 primes (via a gcd computation). It then performs $\min\{t, 256\}$ rounds of Miller-Rabin testing, selecting the bases incrementally from a hard-coded list of the first 256 primes. The code documentation states that when performing $t = 32$ rounds of testing, the probability of a false prime classification is estimated as 2^{-64}.

CommonCrypto provides the primailty test `CCBigNumIsPrime` which calls `ccz_is_prime` provided by corecrypto. However, the user has no choice over the number of rounds of testing t in this case, as it is hard-coded to 16.

Pseudoprimes. Since the bases are chosen deterministically based on the value of t, we can achieve a failure rate of 100% with respect to that value t, simply by using the method of Section 3.2.1.2 to produce a composite n that has the first t primes as non-witnesses. Such an n is in fact guaranteed to be declared prime by `ccz_is_prime`, for any user input $\leq t$. We give examples of composites n that will

be declared prime by corecrypto for $t \leq 256$ and $t \leq 40$ in Section 3.3.17. (These examples are shared between a few different libraries, so are placed in a separate section for clarity.)

Since CommonCrypto forces `ccz_is_prime` to perform 16 rounds of testing, we can achieve a 100% success rate even more easily in this case. Indeed, both examples in Section 3.3.17 are guaranteed to be declared prime in this case, as are the supplementary examples for a variety of bit-sizes provided below.

Example Pseudoprimes for Apple CommonCrypto. Using our SAGE implementation of the method as described in Section 3.2.1.2 with A containing the first 16 primes, we construct three compsites n of the form $n = p_1 p_2 p_3$, where $p_i = k_i(p_1 - 1) + 1$ with $(k_2, k_3) = (113, 173)$ where:

$$p_1 = \texttt{0x32972d4e607a45f57d7144df60a7abf8b473a1680b}$$

produces a 512-bit n,

$$\begin{aligned} p_1 = 2^{192} &\cdot \texttt{0x0000000000151452e0a832f27b9eead0000000000000000} \\ + 2^{0} &\cdot \texttt{0x00000000000000000000000aff3796792e7ceb8d55206a3} \end{aligned}$$

produces a 1024-bit n, and

$$\begin{aligned} p_1 = 2^{576} &\cdot \texttt{0x000000000000000000000032972d4e607a45f57d66c00000} \\ + 2^{0} &\cdot \texttt{0x000000000000000000000001f2a7b5c6a50fc1e38aae911b} \end{aligned}$$

produces a 2048-bit n.

These composites are always declared prime by CommonCrypto.

3.3.6 Cryptlib

Cryptlib 3.4.3 [71] is an open source security toolkit library developed by Peter Gutmann. It provides a variety of services including: public-key algorithms, various cryptographic functions and primality testing.

Analysis. The primality test in Cryptlib is the function `primeProbable` found in `kg_prime.c` and is composed of t rounds of Miller-Rabin, where the value of t must be between 1 and 100 (inclusive) and is chosen by the user upon calling. The function then chooses the base for each test incrementally from the start of a fixed list of primes. This is either a list of the first 54 primes (2 to 251) or the first 2048 primes (2 to 17863), depending on the preprocessor directive `CONFIG_CONSERVE_MEMORY`.

Pseudoprimes. Since $t \leq 100$, we will at most only ever test using the primes between 2 and 541 (the hundredth prime) as bases. We can therefore generate numbers that are guaranteed to be declared prime by this test for any valid input t, simply by using Arnault's method to generate a composite n that has the first 100 primes as non-witnesses. Indeed, using the method described in Section 3.2.1.2 we can produce a 2315-bit composite that is pseudoprime to all prime bases up to and including 541.

An Example Pseudoprime for Cryptlib. Using our SAGE implementation of the method as described in Section 3.2.1.2 with A containing the first 100 primes, we construct a 2315-bit n of the form $n = p_1p_2p_3$, where $p_i = k_i(p_1 - 1) + 1$ with $(k_2, k_3) = (641, 677)$ and

$$\begin{aligned} p_1 = 2^{576} &\cdot \texttt{0x24a027808260908b96d740bef8355ded63f6edb7f70de9a9} \\ + 2^{384} &\cdot \texttt{0xb99c408f131cef3855b4b0aea6b17a4469ed5a7ec8b2be62} \\ + 2^{192} &\cdot \texttt{0x66c3a9eae83a6769e175cb2598256da977b9e191b9b847a7} \\ + 2^{0} &\cdot \texttt{0xe2cf4750d9bc2d64ccd3406f5db662c22c3fc65e3c56eff3}. \end{aligned}$$

This n is declared prime for any valid number of rounds t of testing performed by Cryptlib's primality test.

3.3.7 JavaScript Big Number (JSBN)

The Java Script Big Number (JSBN) library written by Tom Wu [163] provides a small cryptographic toolkit for Java Script applications. Here we study the most recent release JSBN 1.4 from 2013. According to its homepage the library has been

used in a variety of applications, including: Forge (a pure JavaScript implementation of SSL/TLS), Google's V8 benchmark suite version 6, the JavaScript Cryptography Toolkit, and the RSA-Sign JavaScript library.

Analysis. The library offers the primality test `bnIsProbablePrime(t)` where the parameter `t` defines the number of rounds of Miller-Rabin the user wishes to perform. The code documentation states that this function will "test primality with certainty $\geq 1 - .5^t$". The function pseudorandomly chooses a base `a` for each round of Miller-Rabin from a hard-coded list of all primes below 1000 called `lowprimes`.

Pseudoprimes. We can consider this implementation as performing tests with fixed bases, where the bases chosen are all primes between 2 and 1000. We can then use Arnault's method (Section 3.2.1.2) to construct composite numbers n that pass JSBN's primality test no matter how many rounds of testing t the user wishes to perform.

An Example Pseudoprime for JSBN. Using our SAGE implementation of the method as described in Section 3.2.1.2 with A containing the first 1000 primes, we construct a 4279-bit n of the form $n = p_1 p_2 p_3$, where $p_i = k_i(p_1 - 1) + 1$ with $(k_2, k_3) = (1013, 2053)$ and

$$\begin{aligned} p_1 = {} & 2^{1344} \cdot \texttt{0x000000000000000000000000000083dda18eb04a7597ca3} \\ & + 2^{1152} \cdot \texttt{0xc6bc877df8a08eec6725fa0832cba270c42adc358bc0cf50} \\ & + 2^{960} \cdot \texttt{0xc82cb10f2733c3fb8875231fc1498a7b14cb675fac1bf3c5} \\ & + 2^{768} \cdot \texttt{0x127a76fc11e5d20e27940c95ceba671fe1c4232250b74cbd} \\ & + 2^{576} \cdot \texttt{0xf8448c90321513324c0681afb4ba003353b1afb0f1e8b91c} \\ & + 2^{384} \cdot \texttt{0x60af672a5a6f4d06dd0070a4bc74e425f3eae90379e57754} \\ & + 2^{192} \cdot \texttt{0x82d26e80e247464a4bb817dfcf7572f89f8b9cacd059b584} \\ & + 2^{0} \cdot \texttt{0x0e4389c8af84f6a6ea15a3ea5d62cb994b082731ba4cde73}. \end{aligned}$$

This produces an n that is *guaranteed* to be declared prime by JSBN's primality test for *any* certainty parameter t.

3.3.8 LibTomMath

LibTomMath v1.0.1 [45] is an open source multiple-precision integer library with a number theoretic toolkit.

Analysis. LibTomMath includes several methods for primality testing in the form of trial division, Fermat tests, and Miller-Rabin tests. The latter two take a single base a and a number n to test as arguments and return whether a is a witness or non-witness. The main primality test is defined by the function `mp_prime_is_prime`, which takes arguments n (the number to be tested), and integer t with $1 \leq t \leq 256$. It then performs some trial divisions (on a default of the first 256 primes) and then t rounds of Miller-Rabin. The selection of bases to be used is made similarly as in Cryptlib: it simply picks incrementally from a list of hard-coded primes (but this time a list of 256 primes up to 1619 are used).

The documentation of LibTomMath (`bn.pdf`) discusses the number of rounds of Miller-Rabin required with the statement: *"Generally to ensure a number is very likely to be prime you have to perform the Miller-Rabin with at least a half-dozen or so unique bases."* This is complemented with a function `mp_prime_rabin_miller_trials` that gives the number of rounds needed to achieve an error rate less than 2^{-96} based on the bit-size of the number tested (similar to that in OpenSSL and [41]) and a comment in the header file `tommath.h` above `mp_prime_rabin_miller_trials` that states the probability of a false classification is no more than $(1/4)^t$.

Pseudoprimes. Since the bases are chosen deterministically based on the value of t, we can achieve a failure rate of 100% simply by using the method of Section 3.2.1.2 to produce a composite n that has the first 256 primes as non-witnesses; such an n is guaranteed to be declared prime by `mp_prime_is_prime`, for any value of t (including the t chosen by `mp_prime_rabin_miller_trials` that describes an error rate less than 2^{-96}). Section 3.3.17 provides a 7023-bit example of such an n. Much smaller examples can be obtained if smaller values of t are guaranteed to be used; in particular, we can easily obtain a 1024-bit example for $t \leq 40$ (see also Section 3.3.17).

3.3.9 LibTomCrypt

LibTomCrypt v1.18.1 [44] is an additional cryptographic toolkit that shares many resources with LibTomMath.

Analysis. The primality test in LibTomCrypt is called as `isprime(n,t,result)`. It takes as arguments an n to test and carries out t rounds of Miller-Rabin. The documentation of LibTomCrypt advises that each round of Miller-Rabin reduces the probability of n being a pseudoprime by a factor of 4, and therefore deduces that the overall error is at most $(1/4)^t$. LibTomCrypt supports selection from three different big integer libraries at runtime.

If LibTomMath is chosen then `isprime` will call `mp_prime_is_prime` as described in Section 3.3.8, passing on parameters n and t. If TomsFastMath [46] is chosen then `isprime` will call `fp_isprime_ex`, a function defined in the math library TomsFastMath that performs equivalent testing as LibTomMath's `mp_prime_is_prime`. If GMP is selected then `isprime` will call `mpz_probab_prime_p` as described in Section 3.3.2. The value of t used by any of the three choices is inherited from the original call to `isprime`, however if $t = 0$ the value is overwritten to $t = 40$.

Pseudoprimes. If either LibTomMath or TomsFastMath are selected, the pseudoprimes described in Section 3.3.8 will always be declared prime by the primality test, for an example see Section 3.3.17. If GMP is selected we can apply the analysis in Section 3.3.2 to generate pseudoprimes.

3.3.10 WolfSSL

WolfSSL 3.13.0 [160] (formerly CyaSSL) is a small SSL/TLS library targeted for use in embedded systems. WolfSSL provides primality testing tools based on public domain TomsFastMath 0.10 [46] and LibTomMath 0.38 [45] functions.

Analysis. The primality test in WolfSSL is the function `mp_prime_is_prime` which takes a number n to be tested and the rounds of testing t as parameters. This function is directly taken from an older version of LibTomMath, namely 0.38 [45]. WolfSSL will use LibTomMath by default, but can optionally be compiled to use TomsFastMath 0.10 [46] at runtime. The primality test in LibTomMath 0.38 is unchanged from that analysed in version 1.0.1 in Section 3.3.8. When using TomsFastMath, `mp_prime_is_prime` calls `fp_isprime` which strips the user's choice of t and simply calls `fp_isprime_ex` with the hard-coded value of $t = 8$. The function `fp_isprime_ex` then performs trial division (on a default of the first 256 primes) and then does 8 rounds of Miller-Rabin using the first 8 primes as bases. It thus acts equivalently to `mp_prime_is_prime` in LibTomMath, but with $t = 8$.

Pseudoprimes. Since the testing in WolfSSL is in effect the same as that performed in LibTomMath (but using only 8 rounds of Miller-Rabin when using TomsFastMath), the composite examples given in Section 3.3.17 are also declared prime with 100% success.

3.3.11 Libgcrypt

Libgcrypt [87] is a general purpose cryptographic library originally based on code from GnuPG. The library provides various cryptographic functions, including public-key algorithms, large integer functions and primality testing. We analyse version 1.8.2, released in December 2017.

Analysis. The documentation for Libgcrypt states that the function used for checking the primality of primes is `gcry_prime_check` which is found in `primegen.c`. This function then calls `check_prime` in which the actual testing performed. This function `check_prime` performs three testing steps. The first step is trial division with all primes up to 4999. The second step is a Fermat test with base $a = 2$. The last step comprises t rounds of Miller-Rabin where the bases are pseudorandomly chosen. We note that t is user-defined, but cannot be set to less than 5. The default for checking the numbers produced in the prime generation algorithm is set to 5, but when a user calls `gcry_prime_check` the choice of t is hard-coded to 64.

Pseudoprimes. Following Section 3.2.1, beating steps 1 and 2 of the testing performed in `check_prime` is trivial if we choose n as a Carmichael number of the form $n = pqr$ where $p, q, r > 4999$. By using the hybrid technique in Section 3.2.1.4, we can create a Carmichael number that also has the maximum number of randomly distributed non-witnesses. We then need only to overcome the t Miller-Rabin tests with pseudorandom bases. This happens with probability $(1/4)^t$. If the user calls `gcry_prime_check` then the probability with which we can fool this test would be only 2^{-128}. Yet performing 64 rounds of Miller-Rabin is quite time consuming, and a user may be tempted to bypass `gcry_prime_check` and call `check_prime` with fewer rounds. In this hypothetical situation, or in versions of Libgcrypt prior to 1.3.0 (2007) [86] (where `gcry_prime_check` would call $t = 5$ rounds by default) the best we could achieve is passing the test with probability 1/1024 (for $t = 5$).

3.3.12 Java

Java implementations provide their own methods for arbitrary precision arithmetic, including primality tests, as seen in `java.math.BigInteger`. We consider OpenJDK10 [38], although there seems to be no significant changes to this section of the code in older versions such as JDK8.

Analysis. The primality testing function `isProbablePrime` is passed a single parameter `certainty`. This is a value chosen by the user and is described in the documentation as: "a measure of the uncertainty that the caller is willing to tolerate: if the call returns true the probability that this BigInteger is prime *exceeds* $(1 - 1/2^{\texttt{certainty}})$." The `certainty` parameter is then used to determine how many rounds of testing will be performed. This is done by calling the function `primeToCertainty`. We include the source code of the function `primeToCertainty` from the class `java.math.BigInteger`.

This function first sets a variable `n` as $(\texttt{certainty} + 1)/2$. This would produce a non-integer result when `certainty` is even, yet the result is cast to an integer, implicitly flooring the result.[4]

[4]Because of the role that `n` plays in determining the number of rounds of Miller-Rabin to be performed, the result is that there is no difference in testing `isProbablePrime(k)` and

Listing 3.1: `OpenJDK10 java.math.BigInteger` **function** `primeToCertainty`

```
boolean primeToCertainty(int certainty, Random random) {
    int rounds = 0;
    int n = (Math.min(certainty, Integer.MAX_VALUE-1)+1)/2;

    // The relationship between the certainty and the number of rounds
    // we perform is given in the draft standard ANSI X9.80, "PRIME
    // NUMBER GENERATION, PRIMALITY TESTING, AND PRIMALITY CERTIFICATES".
    int sizeInBits = this.bitLength();
    if (sizeInBits < 100) {
        rounds = 50;
        rounds = n < rounds ? n : rounds;
        return passesMillerRabin(rounds, random);
    }

    if (sizeInBits < 256) {
        rounds = 27;
    } else if (sizeInBits < 512) {
        rounds = 15;
    } else if (sizeInBits < 768) {
        rounds = 8;
    } else if (sizeInBits < 1024) {
        rounds = 4;
    } else {
        rounds = 2;
    }
    rounds = n < rounds ? n : rounds;

    return passesMillerRabin(rounds, random) && passesLucasLehmer();
}
```

This function also takes into consideration the bit-size of the number being tested; if it is less than 100, then Miller-Rabin is performed with at most 50 rounds; if it is greater than 100, then both Miller-Rabin and a Lucas probable prime test with Selfridge's parameters are performed, as described in Section 3.2.2. In the latter case, the maximum number of rounds of Miller-Rabin is determined based on the bit-size of the tested number, similarly to OpenSSL. In both cases, the user's choice of `certainty` will determine the actual number of rounds of Miller-Rabin performed only if it is *less* than the internally-specified number for that bit-size.

Pseudoprimes. For numbers of cryptographically interesting size, Java performs both Miller-Rabin and Lucas probable prime tests. Using the method outlined in Section 3.2.2 we could produce composites that are guaranteed to be declared prime by the Lucas test. However, the resulting forms do not fit into any of the known

`isProbablePrime(k+1)` when k is odd. This has an effect on the assurance given to the user — the guarantee of $1 - 1/2^{\texttt{certainty}}$ is no longer accurate for half of the values of `certainty`.

families of composites having high numbers of Miller-Rabin non-witnesses. Hence, we are unable to construct any numbers passing Java's primality test with high probability using our current techniques.

3.3.13 Bouncy Castle

Bouncy Castle is a cryptographic library written in Java and C# [116]. The primality test in Bouncy Castle Java is based on the `BigInteger` class from JDK as described in Section 3.3.12. Bouncy Castle C# implements its own primality tests. We analyse Bouncy Castle C# version 1.8.2.

Analysis. The relevant function responsible for primality tests is located in the class `BigInteger`. This class provides method `IsProbablePrime` which accepts `certainty` as a parameter. The method then uses Miller-Rabin tests with t rounds, where t is computed as $t = ((\texttt{certainty}-1)/2)+1$. In each round the base is selected using a secure random number generator (`SecureRandom`) which is provided by the Bouncy Castle library.

The `certainty` parameter must always be provided to invocation of the `IsProbablePrime` method. Therefore, the user's choice completely determines how many Miller-Rabin rounds are performed. For example, this method is directly used in the `TlsDHUtilities` class, which provides Diffie-Hellman operations for TLS. When validating the incoming DH parameters, the `ValidateDHParameters` method invokes `isProbablePrime` with `certainty` = 2. This results in only a single Miller-Rabin test being carried out.

Pseudoprimes. We can produce composites n using any of the methods in Section 3.2.1; such n meet the Monier-Rabin bound and so will pass Bouncy Castle's primality testing with probability $(1/4)^t$ with t as derived from `certainty`. Although there is no formal documentation, a comment above the primality testing code indicates that the failure rate of this testing function should be $(1/2)^{\texttt{certainty}}$, and so the user's choice of `certainty` *is* achieved.

3.3.14 Botan

Botan is a cryptographic library written in C++11 [94]. In addition to the crypto functionality it offers a TLS client and server implementation. We analyse Botan 2.6.0.

Analysis. The relevant primality test implementation can be found in `numthry.cpp`, which contains function `is_prime`. This function first evaluates whether a tested number is divisible by small primes up to 65521. It then performs Miller-Rabin primality tests with randomly chosen bases. The source of randomness and the number of Miller-Rabin rounds are based on parameters passed to the `is_prime` function.

The number of rounds is computed based on parameter `prob` and t is set as $(\texttt{prob} + 2)/2$. Botan's documentation is very clear on the distinction between testing numbers of random and possibly adversarial origin. To distinguish the source, the function `is_prime` contains a boolean flag `is_random`. If set, then the code uses [41] to assign t based on the bit-size of the number being tested, with a target failure rate less than 2^{-80}.

Pseudoprimes. As with Bouncy Castle, we can produce composite n using any of the methods in Section 3.2.1; such n meet the Monier-Rabin bound and will pass Botan's primality test with the highest probability of $(1/4)^t$ where t is from the user's choice of `prob` via $t = (\texttt{prob} + 2)/2$. In this sense, the test's guarantees match the user's expectations.

3.3.15 Crypto++

Crypto++ 7.0 is an open source C++ cryptography library originally written by Wei Dai [40]. Crypto++ has a variety of primality testing algorithms in `nbtheory.cpp`. These consist of trial division, Fermat, Miller-Rabin and both strong and standard Lucas probable prime tests. Crypto++'s primality testing function `isprime` per-

forms both Miller-Rabin and strong Lucas tests. Thus, to fool it, we would need to find Baillie-PSW pseudoprimes (though the Miller-Rabin test is a random base test, unlike that performed in Baillie-PSW). We do not currently know any such pseudoprimes.

3.3.16 Golang

The Go programming language (Golang) 1.10.3 [62] created at Google in 2009 is an open source project including arbitrary-precision arithmetic and cryptographic functionality.

Analysis. The relevant primality test implementation can be found in `int.go`, which contains function `ProbablyPrime(t)`. The parameter `t` defines the number of rounds of Miller-Rabin the user wishes to perform. The function first performs trial division with a series of small primes, then `t` rounds of Miller-Rabin (where one base is forced to be 2 and all other bases are chosen pseudorandomly), and finally a Lucas probable prime test. Therefore the function is performing a Baillie-PSW test. Before version 1.8, Go's `ProbablyPrime(t)` function applied only the Miller-Rabin tests. The documentation provided by Golang makes it clear that the probability of the function declaring a randomly chosen composite input to be prime is at most $(1/4)^t$. It also states that "`ProbablyPrime(t)` is not suitable for judging primes that an adversary may have crafted to fool the test".

From an attack perspective it is interesting that the pseudorandom number generator used in this primality test is seeded with the tested number n. Thus, an attacker can reliably predict the pseudorandomly generated Miller-Rabin bases.

Pseudoprimes. Since a Baillie-PSW test is being performed, we know of no composites that are incorrectly declared prime by Golang. However, for versions prior to 1.8 released in 2017, we are able to exploit the insecure nature of the Miller-Rabin base selection to produce composite numbers that are guaranteed to be declared prime with respect to a parameter t. Since this is the same method GNU GMP uses to choose bases for Miller-Rabin, we can use the method described in Section 3.3.2

to produce such composites.

We now give an example of a composite n that is always declared prime for $t \leq 13$.

An Example Pseudoprime for Golang pre-1.8. Using the method described in Section 3.3.2, we construct a 1024-bit composite n that is declared prime by Golang's primality test in versions prior to 1.8 with 100% success for $t \leq 13$. We take

$$\begin{aligned} n = 2^{960} &\cdot \texttt{0x00000000000000000000000000000000ff7d428a8a9f9ffc} \\ + 2^{768} &\cdot \texttt{0x2ea178501115ec855f1154c054f5f67e15967a139a92fe15} \\ + 2^{576} &\cdot \texttt{0xddf2c49b044820ea8c58551b74f81b45b116da4e1f11b926} \\ + 2^{384} &\cdot \texttt{0x93e0cdc58006bc2052eb9b2fc32c71dd041d1907225e2814} \\ + 2^{192} &\cdot \texttt{0xebe18736f626fea57c965b67b296a6461455226b39aba263} \\ + 2^{0} &\cdot \texttt{0x3faeb483847a715c6a01d8d0e401a4aaf8f3d22121fd142f}. \end{aligned}$$

3.3.17 Example Pseudoprimes for Apple corecrypto and CommonCrypto, LibTomMath, LibTomCrypt and WolfSSL

Using our SAGE implementation of the method as described in Section 3.2.1.2 with A containing the first 256 primes, we constructed a composite number n that is a pseudoprime to all bases $a \in A$. While the size of this number n may seem arbitrary, it was simply the first number constructed that met all requirements. Previous to this, we are unaware of any pseudoprime constructed by the Arnault method to have as many as 256 chosen bases in A.

We construct a 7023-bit n of the form $n = p_1 p_2 p_3$, where $p_i = k_i(p_1 - 1) + 1$ with

$(k_2, k_3) = (2633, 5881)$ and

$$\begin{aligned}
p_1 = {} & 2^{2304} \cdot \texttt{0x0000000000000000000000000000000000000000001e46d6a8} \\
& + 2^{2112} \cdot \texttt{0x4d42d684ddb3415e871b661303b1c60f0388dfb9e525f8bc} \\
& + 2^{1920} \cdot \texttt{0x51c9de3c9f45627608de2f75dee580d9d4d97cab6fa86dad} \\
& + 2^{1728} \cdot \texttt{0x9e6bbfd721f297472480a9bed9508aa884bda9dc56833752} \\
& + 2^{1536} \cdot \texttt{0xfac8e89f413a9517d14731277148789987806654a8723593} \\
& + 2^{1344} \cdot \texttt{0xa452f960facc9b65f6962cb26131b42650c29c8735083c7e} \\
& + 2^{1152} \cdot \texttt{0x6c3a220d77d1cbe7f9628885a7b79465287d4b02ad546007} \\
& + 2^{960} \cdot \texttt{0x1d43306a8813836de5ccd162fbeca4f117552dba01975451} \\
& + 2^{768} \cdot \texttt{0x2f7684e32b0377e76f87b96906f8fa276381db612f76c2c7} \\
& + 2^{576} \cdot \texttt{0xdd97ab4380042c991a4719884377c70065a3614237a41289} \\
& + 2^{384} \cdot \texttt{0x24a1017fbb529443b0ad43c5424753db5b518cee5a1fcd87} \\
& + 2^{192} \cdot \texttt{0xea038ffcad33380db1d89cd4e0b15b480cf0c62e8999924d} \\
& + 2^{0} \cdot \texttt{0x0284af806081ea106f35f85a664456166b864650ef034cf3}.
\end{aligned}$$

This n is declared prime for any valid number of rounds t for the Apple corecrypto and CommonCrypto, LibTomMath, LibTomCrypt and WolfSSL libraries.

Also using the method as described in Section 3.2.1.2 but now with A containing the first 40 primes, we can construct a 1024-bit n of the form $n = p_1 p_2 p_3$, where $p_i = k_i(p_1 - 1) + 1$ with $(k_2, k_3) = (233, 241)$ and

$$\begin{aligned}
p_1 = {} & 2^{192} \cdot \texttt{0x000000000000e17516504450e648b6aedb0c0784e17dda33} \\
& + 2^{0} \cdot \texttt{0x63e1956a843076a9e5b6d15a819cf0907a96154d47662d0b}.
\end{aligned}$$

This n is guaranteed to be declared prime by `mp_prime_is_prime` with $t \leq 40$, and therefore also guaranteed to be declared prime by `mp_prime_is_prime` as in LibTomCrypt 1.18.1 and WolfSSL 3.13.0 for the same values of t. The same applies for Apple corecrypto and CommonCrypto.

3.4 Mathematical Software

In this section we conduct an analysis of primality testing found in a selection of popular mathematical software. We focus mainly on computer algebra systems that provide number-theoretic tools, as these offer more functionality in primality testing, namely: Magma, Maple, MATLAB, Maxima, SageMath, SymPy and Wolfram Mathematica. Since this software is usually made to run on less computationally constrained devices, with a higher focus on mathematical accuracy than efficiency, we expect the testing here to be more thorough. We include these in our analysis since they might be relied upon by developers when manually checking values in standards or code. Some of the libraries use deterministic tests for proving primality, though most still rely on probabilistic methods when testing candidates larger than 64 bits in size. Maple, Maxima and SymPy have dependencies on GMP and therefore inherit the same issues with its primality test as discussed in Section 3.3.2; however they all also perform Lucas tests in their latest versions, so this "cross contamination" does not result in exploitable weaknesses. Our findings are summarised in Table 3.5.

Table 3.5: Results of our analysis of mathematical software. This shows how the number of rounds of Miller-Rabin used is determined, whether a Baillie-PSW test is implemented, the documented failure rate of the primality test (that is, the probability that it wrongly declares a composite to be prime), and our highest achieved failure rate for composite input.

Software	Rounds of MR	Baillie-PSW?	Documented Failure Rate	Our Highest Failure Rate
Magma V2.23-9[†]	Default 20	No	$(1/4)^t$	$(1/4)^t$
Maple 2017	5	Yes, on $n > 2^{33}$	Not given	0%
MATLAB *R2019b*	10	No	Not given	$(1/4)^{10}$
Maxima 5.41.0	25	No[††]	$(1/4)^{25}$	0%
SageMath 8.2	1	Yes	Not given	0%
SymPy 1.0[‡]	46	No	"small probability"	100%
SymPy 1.1	1	Yes, on $n > 2^{64}$	"small probability"	0%
Wolfram Mathematica 11.3	2	Yes	"correct for $n < 10^{16}$"	0%

[†] When testing input of size at least 34×10^{13} or invoking the test with the parameter `proof = False`.

[††] An additional Lucas test is performed on input of size at least 2^{49}.

[‡] When testing input of size at least 2^{64}.

3.4.1 Magma

Magma V2.23-9 [29] is a mathematical software package designed for computations in algebra, number theory, algebraic geometry and algebraic combinatorics.

Analysis. Magma provides a primality testing function that can either invoke a primality proving algorithm, or what they call a probable-primality test, depending on the arguments given when called. The main function call for primality testing is `IsPrime(n: Proof)`. The more rigorous method of primality proving is based on an implementation of the ECPP (Elliptic Curve Primality Proving) method [10]. It is used by default, unless the number tested is greater than 34×10^{13} or the parameter `Proof` is set to `False`. In this case, the probable-primality test `IsProbablePrime` is instead called. By default, this consists of 20 rounds of Miller-Rabin with random bases. By setting the optional parameter `Bases` to some value B, the number of bases used is B instead of 20.

Pseudoprimes. The pseudoprimes generated in Section 3.2 attempt only to overcome probabilistic primality testing and are not designed to overcome primality proving methods such as ECPP.

However, if the parameters are set to invoke the probable-primality test with default parameters, then composites generated by the methods in Section 3.2.1 have a probability of 2^{-40} of being falsely declared prime. This probability is correctly alluded to as being *worst-case* by the documentation given for this function.

3.4.2 Maple

Maple 2017 [157] is a computer algebra system developed by Maplesoft, that provides a general-purpose software tool for mathematics, data analysis, visualisation, and programming.

Analysis. The primality test in Maple is called as `isprime(n)` on a candidate n to be tested. Documentation states that "*It returns false if n is shown to be composite within one strong pseudo-primality test and one Lucas test. It returns true otherwise*". The function begins with some trial division on a series of small primes before calling `gmp_isprime(n)`. If the result of `gmp_isprime(n)` is 1 (i.e. the number is "probably prime") and the candidate n being tested is greater than $5 \times 10^9 \approx 2^{33}$, then `isprime` will go on to perform a Lucas test on n. In all other cases, the Lucas test is omitted.

Although we cannot directly inspect the code of `gmp_isprime(n)` (since Maple is proprietary software) we are able to reverse-engineer this function by calling it on our own input n and assessing how it performs. Maple's documentation states that it performs a Miller-Rabin test and uses GMP for this function, yet since there is no other code indicative of a Miller-Rabin test in `gmp_isprime(n)`, we deduce that Maple is calling GMP's function `mpz_probab_prime_p(n, reps)`. Since `gmp_isprime(n)` takes only a single argument, we inferred that Maple passes a hardcoded value of `reps` to GMP.

We were able to verify that the value of `reps` is actually 5. We did this by using the methods described in Section 3.3.2 to generate composite numbers of various bit-sizes that are declared prime by `mpz_probab_prime_p (n, reps)` for `reps = 1,2,3,4,5`. For composites that can only pass at most `reps = 4` tests, Maple's `gmp_isprime` correctly identifies these as composite. But for composites that pass `reps = 5`, the function falsely declares them to be prime.

Pseudoprimes. When testing numbers $n \leq 5 \times 10^9$, `isprime` acts as a deterministic version of the Miller-Rabin test. We have verified this by calling `mpz_probab_prime_p(n,5)` for all $n \leq 5 \times 10^9$ and comparing the results to a list of primes below 5×10^9. The different sets of bases that GMP chooses for each n are such that there are no composites below this threshold that are declared prime by `mpz_probab_prime_p` with `reps > 3`. However, any change made to the (flawed) way GMP currently chooses its bases for testing could actually make Maple's `isprime` function *less* accurate (and no longer deterministic) for $n \leq 5 \times 10^9$!

To fool Maple's primality testing for numbers larger than 5×10^9, we would need a composite n passing a Lucas test and 5 rounds of Miller-Rabin testing. We do not currently know any such n.

3.4.3 MATLAB

MATLAB *R2019b* is a widely used numerical computing environment and proprietary programming language developed by MathWorks. It provides capabilities for a variety of mathematical tasks including linear algebra, calculus and number theory.

Analysis. The primality test in MATLAB is called as `isprime(n)` on a candidate n to be tested. The documentation states that `isprime(n)` is performing the Miller-Rabin primality test with 10 independently and randomly chosen bases. MATLAB's documentation does not mention explicit bounds of error probabilities, but does inform users: "*If n is positive and isprime returns TRUE, then n is prime with a very high probability.*".

Like Maple, MATLAB is proprietary software. This means we are unable to directly inspect all parts of the code. However, we can use documentation and tools such as `edit` to inspect some of the code of the inbuilt functions. By calling `edit isprime` we are able to see only the preliminary testing performed on n – this is mainly sanity checking, but also includes MATLAB's trial division code.

Pseudoprimes. If MATLAB is indeed just performing 10 rounds of Miller-Rabin on n, we can produce composite numbers n using any of the methods in Section 3.2.1; such n meet the Monier-Rabin bound and will pass MATLAB's primality test with the highest probability of $(1/4)^{10}$, yet we were unable to verify this claim.

3.4.4 Maxima

Maxima 5.41.0 [98] is a free, open source computer algebra system developed by the Macsyma group. Maxima is a general-purpose system including tools for a

variety of mathematical functions and the manipulation of symbolic and numerical expressions.

Analysis. The primality test supplied by Maxima is the function `primep(n)`. When testing an n less than 341550071728321 ($\approx 2^{49}$) a deterministic version of the Miller-Rabin test is used. This is achieved by calling repeated rounds of Miller-Rabin tests with a set of bases for which it has been verified that no composites are falsely declared prime. These are as defined in [77, 104], and therefore can in general be used to create a deterministic test for numbers less than 2^{64}.

When testing an n bigger than 341550071728321, `primep(n)` performs 25 random base Miller-Rabin tests, then conducts one Lucas test. The source Maxima uses for base selection is then provided by the Maxima random number generator, which is an implementation of Mersenne twister MT 19937 [97].

Maxima's documentation correctly states that "The probability that a non-prime n will pass one Miller-Rabin test is less than 1/4. Using the default value 25 for `primep_number_of_tests`, the probability of n being composite is much smaller that 10^{-15}."

Pseudoprimes. When testing numbers $n < 341550071728321$ ($\approx 2^{49}$) the function `primep(n)` is deterministic, so no pseudoprimes can arise. If instead $n > 341550071728321$, then the combination of Miller-Rabin testing and a Lucas test mean that no pseudoprimes for the test are known.

3.4.5 SageMath

SageMath 8.2 (or simply Sage) is a free Python-based open source mathematics software system originally created by William Stein [148] but now developed by many volunteers. Sage provides a toolkit of mathematical functions in areas such as algebra, combinatorics, numerical mathematics, number theory, and calculus.

Analysis. Although there are many methods one could use to test the primality of a number in Sage, the flagship function is `is_prime(n, proof)` found in `/src/sage/rings/integer.pyx`. If called with the value of `proof` set as `True` (default when starting Sage), the function will perform a provable primality test. If set to `False` it uses a strong pseudo-primality test and instead calls `is_pseudoprime(n)`.

The "provable primality test" called when `proof = True` is the PARI [154] `isprime` function. This then uses a combination of the Baillie-PSW test, Selfridge "$p-1$", and Adleman-Pomerance-Rumely-Cohen-Lenstra (APRCL) test [35]. It is indicated in documentation that this test can be "very slow" when testing a prime that "has more 1000 digits".

The "strong pseudo-primality test" called when `proof = False` is less accurate, but much quicker, and is therefore a likely choice when testing large candidates. The candidates are then tested by PARI's `is_pseudoprime(n)`, which consists of a Baillie-PSW test.

Pseudoprimes. Since a Baillie-PSW test is being performed, we know of no composites that are incorrectly declared prime by SageMath for either boolean value of `proof`.

3.4.6 SymPy

SymPy [151] is a free, open source and widely used symbolic computation Python library that provides computer algebra system capabilities.

Analysis. SymPy provides the primality test `isprime(n)`, which like Maxima, uses select bases to perform a deterministic version of Miller-Rabin when testing candidates $n < 2^{64}$. We shall consider two recent versions of SymPy (SymPy 1.0 and SymPy 1.1) since significant changes to the function `isprime` have been made between these versions.

SymPy 1.0. Prior to release 1.1 in July 2017, the primality test `isprime` provided by SymPy 1.0 first conducted some initial trial division, before performing a deterministic version of the Miller-Rabin test using bases described in [77, 104]. For numbers larger than $\approx 2^{53}$, the test would call additional rounds of Miller-Rabin. In all releases up to and including 0.6.6 of 2009, this would simply perform 8 rounds of Miller-Rabin on the bases $\{2, 3, 5, 7, 11, 13, 17, 19\}$. In version 0.6.7 [150], this was increased to 46 rounds of Miller-Rabin, using the first 46 primes as bases. SymPy's documentation addresses the accuracy of its primality test by vaguely stating that "For $n < 10^{16}$ the answer is accurate; greater n values have a small probability of actually being pseudoprimes.". The test remained fundamentally unchanged until version 1.1 in 2017.

SymPy 1.1 onwards. In July 2017 the function `isprime` was revised. Much like Maxima, SymPy would now perform a deterministic version of the Miller-Rabin test on input less than 2^{64}. This is achieved by calling repeated rounds of Miller-Rabin tests with a set of bases for which it has been verified that no composites are falsely declared prime [77, 104]. For numbers greater than 2^{64}, SymPy would instead perform a Baillie-PSW test as described in Section 2.4.4. SymPy's documentation addresses the accuracy of its primality test by stating that "For n $< 2^{64}$ the answer is definitive; larger n values have a small probability of actually being pseudoprimes."

Pseudoprimes. SymPy 1.0 and all previous versions are vulnerable to composite numbers n generated by the methods in Section 3.2.1.2. These numbers are trivial to construct when the final Miller-Rabin test is based on the first 8 primes, but even after the changes made in version 0.6.7, all versions prior to 1.1 would wrongly declare composites generated in this manner to be prime.

An Example Pseudoprime for SympPy 1.0 and previous versions. Using the method of Section 3.2.1.2, we are able to construct a 1024-bit n of the form $n = p_1p_2p_3$ that is pseudoprime to all bases selected by SymPy in all versions prior

to 1.1. Here $p_i = k_i(p_1 - 1) + 1$ with $(k_2, k_3) = (241, 257)$ and

$$p_1 = 2^{192} \cdot \texttt{0x000000000000f8ae31e07964373e4997647e75fa186dd5e7} + 2^0 \cdot \texttt{0xe42ada869da0b3a333813f8102b1fb5f20623d6543e78a3b}.$$

Since SymPy 1.1 introduced a Baillie-PSW test, we can no longer generate composites that would be declared prime by SymPy.

3.4.7 Wolfram Mathematica

Wolfram Mathematica is a computational software package developed by Wolfram Research that covers scientific, engineering, mathematical, and computing fields. The version we study, Mathematica 11.3 [136], (March 2018) features built-in integration with Wolfram Alpha.

Analysis. Mathematica provides the inbuilt primality test `PrimeQ` that is said to perform two Miller-Rabin tests using bases 2 and 3, combined with a "Lucas pseudoprime" test. Since the source code is not open source, we are unable to verify the parameters used in the Lucas test. We note that the documentation references Baillie and Wagstaff [15], from which Selfridge's parameters originate. Documentation of the function also indicates that this procedure is only known to be correct for $n < 10^{16}$ and that "*it is conceivable that for larger n it could claim a composite number to be prime*".

Pseudoprimes. Since a Baillie-PSW test is being performed, we know of no composites that are incorrectly declared prime.

3.5 Application to Diffie-Hellman

Validating the correctness of Diffie-Hellman (DH) parameters is a vital step for verifying the integrity of the key exchange. As mentioned in the introduction of this section, since the DH parameter set (p, q, g), with $g \in \mathbb{Z}_p$ generating a group

of order q, is public, they can originate from third-party sources such as a server or a standard. An adept DH parameter validation function should check that p, q are both prime and that $p = kq+1$ for some integer k. It should also test that the given generator g generates the subgroup of order q and that any received DH values lie in the correct subgroup. A common choice is to set $k = 2$, so that p is a safe prime. For p that are not safe primes, the group order q can be much smaller than p, offering performance improvements. The security level is then based upon the bit-size of q, which must still be large enough to thwart the Pohlig-Hellman algorithm for solving the Discrete Logarithm Problem (DLP), which for prime q runs in time $O(\sqrt{q})$. A common parameter choice is a 160-bit q with a 1024-bit p or a 256-bit q with a 2048-bit p.

More precisely, the Pohlig-Hellman algorithm runs in time $O(\sqrt{t})$ where t is the largest prime factor of q. Thus, an attacker armed with the ability to fool a primality test can supply a sufficiently smooth composite q such that $p = kq+1$ is still prime. For example, if q is of the form $(2x+1)(4x+1)$ this leads to an attack on DLP with complexity 2^{40} resp. 2^{64} for the sizes mentioned above.

We stress, though, that none of the constructions for malicious composites in this chapter pose a risk to protocols such as Telegram that insist on $k = 2$, i.e. which check both $q = (p-1)/2$ and p for compositeness. For example, the construction of Section 3.2.1.1 would set $q = (2x+1)(4x+1)$ and yield p that is always divisible by 3; moreover, q would not be smooth enough for Pohlig-Hellman to pose a threat for parameters of cryptographically appropriate size. However, it is the focus of the next chapter to find a large, sufficiently smooth composite q passing a primality test with a high probability such that $p = 2q+1$ is prime or passes a primality test, too.

We now discuss DH verification functions in various libraries. For each library, we apply the analysis from Section 3.3 to check how robust these libraries are to attack. We note that the other libraries discussed in Section 3.3 do not implement a higher-level function for verification of DH parameters. Of course, this does not prevent an application from using these libraries to realise its own verification function. Such an application would inherit the weaknesses and strengths of the underlying library (when $k \neq 2$ is permitted). We give an example of this scenario for the GMP library below. We close with a discussion of the important use case of SSL/TLS.

OpenSSL. The file `dh_check.c` contains the functions `DH_check_params` and `DH_check`. The former is a lightweight check that just confirms that p and g are 'likely enough' to be valid, by testing to see if p is odd and $1 < g < p - 1$. The latter function is more thorough and calls `BN_is_prime_ex` to test the primality of both p and $q = (p-1)/2$. These primality tests are called with `checks = BN_prime_checks`, therefore the rounds of Miller-Rabin are determined by Table 3.4. This means for example that they will declare as prime with probability $1/16$ composites n of the special form $n = (2x + 1)(4x + 1)$, for x odd and $2x + 1, 4x + 1$ prime, when n has more than 1300 bits. Since no private data is required, this testing function's most likely use-case is checking Diffie-Hellman parameters that have been generated by someone else (perhaps from an untrusted server or an unknown origin) and therefore clearly misuses OpenSSL's own primality testing functions.

In Chapter 4 we exploit the misuse of the average-case error estimates in OpenSSL's Diffie-Hellman parameter checking function to generate parameters (p, q, g) that pass primality testing on both p and q (with some probability) simultaneously and allow efficient solving of the DLP.

Bouncy Castle. The validation of DH parameters in `ValidateDHParameters` extracts p, g from a DH parameter set and then only checks the primality of p with 1 round of Miller-Rabin. We can therefore produce composites that are accepted as DH moduli with probability $1/4$. More seriously, q is not given to the check function, so even with a prime p, the value of g can be chosen so that it has small order, making Pohlig-Hellman as easy as desired. Even if g had large prime order, the flexibility in choosing parameters would allow Lim-Lee small subgroup attacks, as explored in [156].

Botan. The Botan function `is_prime` is used in the class `DL_Group` (located in `dl_group.cpp`) which is also used for verifying DH parameters. This class contains the `verify_group` function, which can be invoked with boolean parameter `strong`. If `strong` is set to `true`, the `is_prime` function is invoked with `prob=128`. This results in $t = 65$ Miller-Rabin computations. Otherwise, `prob=10` and 6 Miller-Rabin computations are performed. This test is performed for both p and q; the code also checks that $q|(p - 1)$ but does not insist on p being a safe prime.

Using the methods described in Section 3.2.1 we can find a q of 160-bits that passes 6 rounds of MR testing with probability $1/4096$ such that q has 2 or 3 prime factors. Then we can construct 1024-bit prime p as $p = kq + 1$ by using the flexibility in k, and a g that generates the subgroup of size q. Since this p is indeed prime and $q|(p-1)$, all of Botan's tests on the parameter set (p, q, g) will pass with probability $1/4096$ if `strong` is set to `false`. We can subsequently use the Pohlig-Hellman algorithm to solve the DLP in the subgroup generated by g and break DH with about 2^{28} effort.

An Example of a Malicious DH Parameter Set for Botan. Using our SAGE implementation of the method in Section 3.2.1.4, we construct a 160-bit q of the form $q = q_1q_2q_3$, where $q_i = k_i(q_1 - 1) + 1$ with $(k_2, k_3) = (61, 101)$ and $q_1 = 537242417098003$.

This q is declared prime with probability $1/4096$ by Botan's `verify_group` function. By setting $k = 2^{864} + 134$ in $p = kq + 1$ we obtain a prime p, and thus by setting the generator g as:

$$\begin{aligned} g = 2^{960} &\cdot \texttt{0x000000000000000000000000000000075ead4f9fa60a06e} \\ + 2^{768} &\cdot \texttt{0x0787a1e0708f5e2055b2899691f7dd73303d5643e57b1636} \\ + 2^{576} &\cdot \texttt{0x66ce328086bd6a0df756175c35549ba7a5ffe517036c0ef1} \\ + 2^{384} &\cdot \texttt{0x44a9542f698255efb66cda28b0b8a5ebebf2c0892f8147d3} \\ + 2^{192} &\cdot \texttt{0x72083822a36098addcd30a1767ccefaae65d1dcd6b45de92} \\ + 2^{0} &\cdot \texttt{0x09047326d40b622af6a76218664ba3df13eb0fead02d772a} \end{aligned}$$

we obtain a parameter set (p, q, g) such that g generates the subgroup of order q. The probability that this set is accepted by Botan's `verify_group` function is $1/4096$. The DLP in the subgroup generated by g can be solved using the Pohlig-Hellman algorithm over each of the 49-bit, 55-bit and 56-bit factors q_1, q_2 and q_3 of q. The cost is dominated by the largest prime factor, and is approximately 2^{28} operations.

GNU GMP. The 256-bit integer $q = (2x + 1)(4x + 1)$ with

$$x = \texttt{0x400286bac15132db85b1c936709f369b}$$

passes 15 rounds of GMP's primality test `mpz_is_probab_prime_p`; picking $k = 2^{1792} + 1254$ produces the 2048 bit prime $p = kq + 1$. The resulting parameter set (p, q, g) would pass even fully adept DH validation with certainty if the underlying primality testing was based on GNU GMP's code with the minimum recommended number of rounds of Miller-Rabin.

SSL/TLS. We close by commenting on the situation for DH parameter testing in SSL/TLS. Here, the server chooses parameters but only sends (p, g) to the client. There is no requirement that p be a safe prime. This makes it difficult for clients to validate the DH parameters (they would need to factor $p - 1$ and then try different divisors to determine the order of q) or to perform group membership tests on received DH values. Consequently most clients perform only simple sanity checks, e.g. checking that $g \notin \{0, \pm 1\}$. This makes SSL/TLS vulnerable to a variety of malicious DH parameter attacks, cf. [161, 156], and in view of these, exhibiting composite p that fool primality tests would be overkill for the SSL/TLS standards in their present form. However, our work shows that even if clients tried to validate DH parameters by factoring $p - 1$, finding the order of g and then testing it for primality, they could still fall foul of malicious DH parameters. And if the SSL/TLS protocol were amended so that the server provides full DH parameters, careful checks would still be needed. Finally we note that only a small number of fixed, safe prime DH parameter sets are permitted in TLS 1.3. These were recently standardised in RFC 7919 [61], alleviating these issues for future versions of the protocol.

3.6 Disclosure and Mitigations

We reported our findings and suggested suitable mitigations based on the outcome of our analysis to OpenSSL, GMP, Mozilla, Apple, Cryptlib, JSBN, LibTomMath, LibTomCrypt, WolfSSL, Bouncy Castle and Botan. We give a short review of the outcome of these discussions.

- When we reached out to the OpenSSL developers, they were in the process of amending its primality testing code to make it FIPS-complaint [117]. These changes were included in OpenSSL versions 1.1.1-pre9, 1.0.2p and 1.1.0i (released 14/08/2018).[5] However, these changes do not consider the adversarial scenario on which our work focuses, and the default settings in OpenSSL remain weak in that scenario. We continued to work with OpenSSL to address this further, and will discuss this throughout Chapters 4 and 5.

- Apple changed its corecrypto library from using fixed bases in Miller-Rabin testing to using pseudorandom bases. This vulnerability was assigned CVE-2018-4398.[6]

- LibTomMath and LibTomCrypt developers are in the process of adjusting the primality testing functions within their library. They plan to remove the fixed base Miller-Rabin testing and replace the function with a Baillie-PSW test in accordance with our recommendations [91].

- WolfSSL has made several adaptations to its primality testing in version 3.15.5 in response to our findings [159]. This includes now performing Miller-Rabin with pseudorandom bases, not overriding the users choice of iterations, and increasing the number of rounds performed on prime parameters in DH and DSA check functions.

- Bouncy Castle has also made changes based upon our findings, by removing the DH verification function and replacing it with a whitelisting approach in release 1.8.3. They are also looking into performing Baillie-PSW in future versions as per our suggestion, yet this is not present in the current release 1.8.6 February 2020.

- Botan version 2.7.0 [95] has increased the number of rounds of Miller-Rabin performed in DH verification and includes the Lucas test to perform Baillie-PSW as per our suggestions.

- Mozilla filed a sec-moderate security bug in response to our disclosure to the primality test in NSS in October 2019[7]. NSS will update the primality test

[5] See `https://www.openssl.org/news/changelog.html`.

[6] See `https://nvd.nist.gov/vuln/detail/CVE-2018-4398` and `https://support.apple.com/en-gb/HT201222`.

[7] See `https://bugzilla.mozilla.org/show_bug.cgi?id=1602379` (requires approval process).

to perform testing on random bases chosen by a more secure PRNG. We have been told this will take place in one of the next NSS releases.

- GNU GMP, Mini-GMP and Cryptlib all remain unchanged, but the authors of Cryptlib pointed out a code comment that indicates the limitations of their primality test.
- We received no response from JSBN.

3.7 Conclusion and Recommendations

Our work has explored primality testing in the adversarial setting and its impact on Diffie-Hellman parameter testing. Our main finding is that leading libraries are not designed for this setting, and therefore often vulnerable to accepting as prime composite inputs that are maliciously chosen, see Table 3.3.

The need for careful distinction between non-adversarial (or random) and adversarial primality testing is of course well understood in the cryptographic research community. However, this distinction is not necessarily reflected and implemented in cryptographic libraries and their documentation. As such, we can generally classify the underlying cause of the failure in prime classification accuracy as a non-consideration of the adversarial setting. More explicitly, we can categorise most failures in terms of how the bases for Miller-Rabin are chosen, i.e. fixed base, predictable bases, insufficient number of bases. Mini-GMP, JSBN, Cryptlib, LibTomMath, LibTomCrypt and WolfSSL all fail due to the selection of bases from a fixed list, whereas GNU GMP and Golang pre 1.8 both suffer from predictable bases. OpenSSL, Libgcrypt, Botan and Bouncy Castle C# all have options to run as many rounds of Miller-Rabin as the user desires, but either default to, or call the test (elsewhere in the library) with too few rounds.

Based on our analysis, we make the following recommendations:

- Libraries that wish to continue to use Miller-Rabin only (for example, to maintain a small codebase) should use pseudorandom bases, cf. Apple corecrypto

and CommonCrypto, Cryptlib, JavaScript Big Number, LibTomCrypt, NSS, WolfSSL. In particular, the bases should not depend only on n, cf. GNU GMP.

- We also recommend to default to worst-case bounds when picking the number of iterations and only assume average-case behaviour when explicitly instructed to by the user. Specifically, we recommend using 64 iterations to ensure that composite numbers are wrongly identified as being prime with probability at most 2^{-128}. The impact on the performance of defaulting to worst-case bounds should be minimal in the non-adversarial setting since testing can be aborted as soon as a Miller-Rabin witness for compositeness is identified, and these are exceedingly common (as the bounds of [41] show). On the other hand, it is precisely in the adversarial setting that the worst-case bounds are needed. Adopting this recommendation may require changes to interfaces to primality testing code. In Chapter 5 we look in more detail at these changes, and perform a cost-benefit analysis for performing Miller-Rabin with 64 rounds.

- If the size and complexity of the codebase is not a concern, or in mathematical libraries for which the functionality already exists (e.g. the computation of Jacobi symbols), it may be suitable to perform the Baillie-PSW test. The negative impact on performance is moderate, and the positive impact on security is significant. An existing benchmark for such a trade-off is found in the documentation of the computer algebra system PARI/GP [154] (on which Sage bases its primality testing functions). PARI/GP implements both a Miller-Rabin test with user-defined t and a Baillie-PSW test and indicate [153] that their Baillie-PSW test is about as fast as their Miller-Rabin test with $t = 3$. In Chapter 5 we will give a more detailed comparison between Baillie-PSW and Miller-Rabin, to better solidify these benchmarks.

- Designers of new protocols should avoid the pitfalls made in SSL/TLS, where DH parameter validation is made impractical for clients. TLS 1.3 does so by fixing and requiring the use of a small collection of parameter sets.

Definitions in the cryptographic literature routinely start with "Let p be a prime ..." whereas our work highlights that many implementations do not necessarily provide strong guarantees for this assumption to hold. It is thus an interesting open question which other seemingly innocuous assumptions concerning domain parameters in the literature can be undermined in a similar fashion.

Chapter 4

Great Exponentiations: On the Need for Robust Diffie-Hellman Parameter Validation

Contents

In this chapter we consider the problem of constructing Diffie-Hellman (DH) parameters which pass standard approaches to parameter validation but for which the Discrete Logarithm Problem (DLP) is relatively easy to solve. We consider both the finite field setting and the elliptic curve setting.

For finite fields, we show how to construct DH parameters (p, q, g) for the safe prime setting in which $p = 2q + 1$ is prime, q is relatively smooth but fools random-base Miller-Rabin primality testing with some reasonable probability, and g is of order q mod p. This problem was left open in Chapter 3. The construction involves modifying and combining known methods for obtaining Carmichael numbers. Concretely, we provide an example with 1024-bit p which passes OpenSSL's Diffie-Hellman validation procedure with probability 2^{-24} (for versions of OpenSSL prior to 1.1.1pre9,

1.1.0i and 1.0.2p Aug. 2018). Here, the largest factor of q has 121 bits, meaning that the DLP can be solved with about 2^{64} effort using the Pohlig-Hellman algorithm. We go on to explain how this parameter set can be used to mount offline dictionary attacks against PAKE protocols.

In the elliptic curve case, we use an algorithm of Bröker and Stevenhagen to construct an elliptic curve E over a finite field $\mathbb{F}_p$ having a specified number of points n. We are able to select n of the form $h \cdot q$ such that h is a small co-factor, q is relatively smooth but fools random-base Miller-Rabin primality testing with some reasonable probability, and E has a point of order q. Concretely, we provide example curves at the 128-bit security level with $h = 1$, where q passes a single random-base Miller-Rabin primality test with probability $1/4$ and where the elliptic curve DLP can be solved with about 2^{44} effort. Alternatively, we can pass the test with probability $1/8$ and solve the elliptic curve DLP with about $2^{35.5}$ effort. These ECDH parameter sets lead to similar attacks on PAKE protocols relying on elliptic curves.

Our work in this chapter shows the importance of performing proper (EC)DH parameter validation in cryptographic implementations and/or the wisdom of relying on standardised parameter sets of known provenance.

4.1 Introduction and Motivation

In Chapter 3 we conducted a systematic study of primality testing "in the wild". We found flaws in primality tests implemented in several cryptographic libraries, for example a reliance on fixed-base Miller-Rabin primality testing, or the use of too few rounds of the Miller-Rabin test when testing numbers of unknown provenance. We also studied the implications of this work for Diffie-Hellman (DH) in the finite field case, showing how to generate DH parameter sets of the form (p, q, g) in which $p = kq + 1$ for some k, p is prime, q is composite but passes a Miller-Rabin primality test with some probability, yet q is sufficiently smooth that the Discrete Logarithm Problem (DLP) is relatively easy to solve using the Pohlig-Hellman algorithm in the order q subgroup generated by g. Such a parameter set (p, q, g) might pass DH parameter validation with non-negligible probability in a cryptographic library that performs "naive" primality testing on p and q, e.g. one carrying out just a few

iterations of Miller-Rabin on each number. If such a parameter set were used in a cryptographic protocol like TLS, then it would also allow an attacker to recover all the keying material and thence break the protocol's security, cf. [161]. In Section 3.5 we posited this as a plausible attack scenario when, for example, a malicious developer hard-codes the DH parameters into the protocol.

It is notable that the methods described in Section 3.2 for producing malicious DH parameters and the examples given in Section 3.5 do not work in the safe prime setting, wherein $p = 2q + 1$. This is because we need flexibility in choosing k to arrange p to be prime. It is also because our methods can only produce q with 2 or 3 prime factors, meaning that q needs to be relatively small so that the Pohlig-Hellman algorithm applies (recall that Pohlig-Hellman runs in time $O(B^{1/2})$ where B is a bound on the largest prime factor of q; if q has only 3 prime factors and we want an algorithm requiring 2^{64} effort, then q can have at most 384 bits). Yet requiring safe primes is quite common for DH in the finite field setting. This is because it helps to avoid known attacks, such as small subgroup attacks [92, 156], and because it ostensibly makes parameter validation easier. For example, OpenSSL's Diffie-Hellman validation routine `DH_check`[1] insists on the safe prime setting by default. Indeed, it was left as an open problem in Chapter 3 to find a large, sufficiently smooth, composite q passing a primality test with high probability such that $p = 2q + 1$ is also prime or passes a primality test.

Interestingly, more than a decade ago, Bleichenbacher [26] addressed a closely related problem: the construction of malicious DH parameters (p, q, g) for which p and q pass *fixed-base* Miller-Rabin primality tests. This was motivated by his observation that, at this time, the GNU Crypto library was using such a test, with the bases being the first 13 primes $a = 2, 3, \ldots, 41$. He produced a number q having 1095 bits and 27 prime factors, the largest of which has 53 bits, such that q *always* passed the primality test of GNU Crypto, and such that $p = 2q + 1$ is prime. His q has very special form: it is a Carmichael number obtained using a modified version of the Erdös method [51]. Of course, his DH parameter set (p, q, g) would not stand up to the more commonly used random-base Miller-Rabin testing, but his construction is nevertheless impressive. Bleichenbacher also showed how such a parameter set

[1] See `https://www.openssl.org/docs/man1.1.1/man3/DH_check.html` for a description and `https://github.com/openssl/openssl/blob/master/crypto/dh/dh_check.c` for source code.

could be used to break the SRP Password Authenticated Key Exchange (PAKE) protocol: he showed that a client that accepts bad DH parameters in the SRP protocol can be subject to an offline dictionary attack on its password. Here, the attacker impersonates the server in a run of the SRP protocol, sending the client malicious DH parameters, and inducing the client to send a password-dependent protocol message. It is the attacker's ability to solve the DLP that then enables the offline password recovery. Thus Bleichenbacher had already given a realistic and quite standard attack scenario where robust DH (and ECDH) parameter validation is crucial: PAKE protocols in which an attacker impersonating one of the parties can dictate (EC)DH parameters.

4.1.1 Contributions & Outline

In this chapter, we address the problem left open from Chapter 3 of finding malicious DH parameters in the safe prime setting. We also study the analogous problem in the elliptic curve setting.

Finite Field Setting: As a flavour of the results to come, we exhibit a DH parameter set $(p = 2q + 1, q, g)$ in which p has 1024 bits and q is a composite with 9 prime factors, each at most 121 bits in size, which passes a single random-base Miller-Rabin test with probability 2^{-8}. We show that no number with this many factors can achieve a higher passing probability. Because of the 121-bit bound on the factors of q, the DLP in the subgroup of order q generated by g can be solved with about 2^{64} effort using the Pohlig-Hellman algorithm. When OpenSSL's DH validation routine `DH_check` is used in its default configuration, this parameter set is declared valid with probability 2^{-24} for versions of OpenSSL prior to 1.1.1pre9, 1.1.0i and 1.0.2p (released 14th August 2018). This is because OpenSSL uses the size of q to determine how many rounds of Miller-Rabin to apply, and adopts non-adversarial bounds suitable for average case primality testing derived from [41]. These dictate using 3 rounds of testing for 1023-bit q for versions of OpenSSL prior to 1.1.1pre9, 1.1.0i and 1.0.2p, and 5 rounds in later versions (the increase was made in an effort to achieve a 128-bit security level). We also give a DH parameter set $(p = 2q+1, q, g)$ in which p is a 1024 bit prime and q has 11 prime factors, each at most 100 bits

in size, which passes a single random-base Miller-Rabin test with probability 2^{-10}. This parameter set is declared valid with a lower probability of 2^{-30} for versions of OpenSSL prior to 1.1.1pre9, 1.1.0i and 1.0.2p, however the DLP in the subgroup of order q generated by g can be solved using the Pohlig-Hellman algorithm with less effort, in about 2^{54} operations.

The probability of 2^{-24} or 2^{-30} for passing DH validation may not seem very large, and indeed can be seen as a vindication of using safe primes for DH. On the other hand, Bleichenbacher-style attacks against PAKEs can be carried out over many sessions and against multiple users, meaning that the success probability of an overall password recovery attack can be boosted. We exemplify this in the context of J-PAKE, a particular PAKE protocol that was supported in OpenSSL until recently (but we stress that the attack is not specific to J-PAKE).

Obtaining such malicious DH parameter sets in the finite field setting requires some new insights. In particular, we are interested in numbers q that are relatively smooth (having all prime factors less than some pre-determined bound B, say), but which pass random-base Miller-Rabin primality tests with probability as high as possible. We therefore investigate the relationship between the number of prime factors m of a number n and the number of Miller-Rabin non-witnesses $S(n)$ for n, this being the number of bases a for which the Miller-Rabin test fails to declare n composite. We are able to prove that $S(n) \leq \varphi(n)/2^{m-1}$ where $\varphi(\cdot)$ is the Euler function. Since for large n we usually have $\varphi(n) \approx n$, this shows that the highest probability a malicious actor can achieve for passing a single, random-base Miller-Rabin test is (roughly) 2^{1-m}. (This already shows that an adversary can only have limited success, especially if multiple rounds of Miller-Rabin are used.) We are also able to completely characterise those numbers achieving equality in this bound for $m \geq 3$: they are exactly the Carmichael numbers having m prime factors that are all congruent to 3 mod 4.

This characterisation then motivates us to develop constructions for such Carmichael numbers with a controlled number of prime factors. We show how to modify the existing Erdös method [51] and the method of Granville and Pomerance [67] for constructing Carmichael numbers, and how to combine them, to obtain cryptographically-sized q with the required properties.

However, this only partly solves our problem, since we also require that $p = 2q + 1$ should pass primality tests (or even be prime). We explore further modifications of our approach so as to avoid trivial arithmetic conditions that prevent p from being prime (the prime 3 is particularly troublesome in this regard). We are also able to show that the probability that p is prime is higher than would be expected for a random choice of p by virtue of properties of the Granville-Pomerance construction: essentially, the construction ensures that p cannot be divisible by certain small primes; we tweak the construction further to enhance this property. Combining all of these steps leads to a detailed procedure by which our example DH parameter set $(p = 2q + 1, q, g)$ described above was obtained. This procedure is amenable to parallelisation. The computation of our particular example required 136 core-days of computation using a server with 3.2GHz processors.

Elliptic Curve Setting: While the main focus of this chapter is on the finite field setting, we also briefly study the elliptic curve setting. Here ECDH parameters (p, E, P, q, h) consist of a prime p defining a field (we focus on prime fields, $\mathbb{F}_p$), a curve E over that field defined in some standard form (for example, short Weierstrass form), a point P, the (claimed) order q of P, and a co-factor h such that $\#E(\mathbb{F}_p) = h \cdot q$. Parameter validation should verify the primality of p and q, and check that P does have order q on E by computing $[q]P$ and comparing the result to the point at infinity.

Bröker and Stevenhagen [32] gave a reasonably efficient algorithm to construct an elliptic curve E over a prime field $\mathbb{F}_p$ having a specified number of points n, given the factorisation of n as an input. Their algorithm is sensitive to the number of prime factors of n – fewer is better. We use their algorithm with n being one of our specially constructed Carmichael numbers q passing Miller-Rabin primality testing with highest possible probability, or a small multiple of such a q.

Since $p \approx q$ in the elliptic curve setting and we only need these numbers to have, say, 256 bits to achieve a 128-bit security level, the task of constructing q is much easier than in the finite field setting considered above. Indeed, we could employ a Carmichael number q with 3 prime factors to pass Miller-Rabin with probability 1/4 per iteration. At the 128-bit security level, q then has 3 prime factors each of

roughly 85 bits, and the Pohlig-Hellman algorithm would solve the ECDLP on the constructed curve in about 2^{44} steps. Using a Carmichael q with 4 prime factors each of exactly 64 bits, we would pass Miller-Rabin with probability 1/8 per iteration and solve the ECDLP with only 2^{34} effort. We give concrete examples of curves having such properties.

These malicious ECDH parameters (p, E, P, q, h) lead to attacks on PAKEs running over elliptic curves, as well as more traditional ECDH key exchanges. These attacks are fully analogous to those in the finite field setting. They highlight the importance of careful validation of ECDH parameters that may originate from potentially malicious sources, especially in the case of bespoke parameter sets sent as part of a cryptographic protocol. For example, the specification of the TLS extension for elliptic curve cryptography [25] caters for the use of custom elliptic curves, though this option does not seem to be widely supported in implementations at present. Our work shows that robust checking of any such parameters would be highly advisable.

4.1.2 Related Work

In the light of the Snowden revelations, a body of work examining methods by which the security of cryptographic algorithms and protocols can be deliberately undermined has been developed. Our work can be seen as fitting into that theme (though we stress that the application of our work to PAKE protocols shows that there are concerns in the "standard" cryptographic setting too).

Young and Yung laid the foundations of kleptography, that is, cryptography designed with malicious intent, see for example [166]. Bellare *et al.* [18] studied the problem of how to subvert symmetric encryption algorithms, and how to protect against such subversions.

Fried *et al.* [54] followed up on early work of Gordon [63] to examine how to backdoor the DLP in the finite field setting. These works showed how to construct large primes p for which the Special Number Field Sieve makes solving the DLP possible if one is in possession of trapdoor information about how p was generated. This provides another avenue to subverting the security of DH parameters. It appears that the

1024-bit example in [54] is not in the safe-prime setting, however.

The NIST DualEC generator was extensively analysed [34] and found to be used in Juniper's ScreenOS operating system in an exploitable way [33]. This inspired more theoretical follow-up work on backdoored RNGs [49] and PRNGs [42].

Bernstein *et al.* [20] extensively discuss the problem of certifying that elliptic curve parameter sets are free of manipulation during generation.

The dangers of allowing support for old algorithms and protocol versions, especially those allowing export-grade cryptography, are made manifest by the FREAK [23], Logjam [3] and DROWN [12] attacks on SSL and TLS.

4.2 Miller-Rabin Primality Testing and Pseudoprimes

In this section, we extend the analysis of Section 3.2.1 on Miller-Rabin pseudoprimes, focusing more on the number of non-witnesses a particular composite number contains. We briefly recap on some of the definitions discussed in the preliminaries in Section 2.4.2 in order to establish and refresh notation.

Suppose $n > 1$ is an odd integer to be tested for primality. We first write $n = 2^e d+1$ where d is odd. If n is prime, then for any integer a with $1 \leq a < n$, we have:

$$a^d = 1 \mod n \text{ or } a^{2^i d} = -1 \bmod n \text{ for some } 0 \leq i < e.$$

The Miller-Rabin test then consists of checking the above conditions for some value a, declaring a number to be composite if both conditions fail and to be (probably) prime if either of the two conditions hold. If n is composite but one condition holds, then we say n is a *pseudoprime to base* a, or that a is a *non-witness to the compositeness of* n (since n may be composite, but a does not demonstrate this fact).

We begin by exploring the relationship between a composite number n and the number of non-witnesses this number possesses, denoted $S(n)$. Since in this work

we are interested in constructing numbers n that fool the Miller-Rabin test with as high a probability as possible for random bases a, our main interest is in constructing n for which $S(n)$ is as large as possible. However, since we are also interested in solving discrete logarithm problems in subgroups of order n, we will also want n to be relatively smooth.

Recall Theorem 2.6 that can be used to calculate the exact number of non-witnesses that some composite n has, and the general upper-bound on $S(n)$ given by results of [108, 135]:

Theorem 4.1 (Monier-Rabin Theorem)**.** *Let $n \neq 9$ be odd and composite. Then*

$$S(n) \leq \frac{\varphi(n)}{4}$$

where φ denotes the Euler totient function.

It is known from [108] that the bound in Theorem 4.1 is met with equality for numbers n of the form $n = (2x+1)(4x+1)$ with $2x+1, 4x+1$ prime and x odd. It is also known that the bound is met with equality for numbers n that are Carmichael numbers with three prime factors, $n = p_1p_2p_3$, and where each factor p_i is congruent to 3 mod 4.

Definition 4.1 (Carmichael numbers)**.** *Let n be an odd composite number. Then n is said to be a Carmichael number if $a^{n-1} = 1 \bmod n$ for all a co-prime to n.*

Note that Carmichael numbers are those for which the Fermat primality test fails to identify n as composite for all co-prime bases a.

Theorem 4.2 (Korselt's Criterion)**.** *Let n be odd and composite. Then n is a Carmichael number if and only if n is square-free and for all prime divisors p of n, we have $p-1 \mid n-1$.*

For a proof of this theorem, see [108]. It is also known that Carmichael numbers must have at least 3 distinct prime factors.

4.2.1 On the Relationship Between $S(n)$ and m, the Number of Prime Factors of n

The Monier-Rabin bound is synonymous with understanding the accuracy of the Miller-Rabin test; it states that any odd composite $n \neq 9$ can have at most $\varphi(n)/4$ non-witnesses, and therefore can pass a single round of Miller-Rabin with probability at most $\approx 1/4$. We present an extension of this bound to classify the maximum number of non-witnesses an odd composite number n can have, with respect to the number m of distinct prime factors it has. We show that is bound is at most $\varphi(n)/2^{m-1}$ non-witnesses, and therefore n has the probability of at most $\approx 1/2^{m-1}$ of being declared prime by a single round of Miller-Rabin. This result is central to our work in this chapter.

Theorem 4.3 (Extended Monier-Rabin Bound)**.** *Let n be an odd composite integer with prime factorisation $n = \prod_{i=1}^{m} p_i^{q_i}$. Write $n = 2^e d + 1$ where d is odd and $p_i = 2^{e_i} d_i + 1$ where each d_i is odd. If $m = 2$, and $n \neq 9$ then,*

$$S(n) \leq \frac{\varphi(n)}{4}.$$

where $\varphi(\cdot)$ denotes Euler's function. Otherwise,

$$S(n) \leq \frac{\varphi(n)}{2^{m-1}},$$

with equality if and only if n is square-free and, for all i, $e_i = 1$ and $d_i \mid d$.

Proof. For $m = 2$ and $n \neq 9$ see the Monier-Rabin Theorem. Otherwise, we have:

$$\begin{aligned}
\frac{\frac{2^{\min(e_i)\cdot m}-1}{2^m-1}+1}{2^{\min(e_i)\cdot m}} &= \frac{1}{2^m-1} + \left(\frac{1}{2^{\min(e_i)\cdot m}}\right)\left(1 - \frac{1}{2^m-1}\right) \\
&\leq \frac{1}{2^m-1} + \left(\frac{1}{2^m}\right)\left(1 - \frac{1}{2^m-1}\right) \\
&= \frac{2(2^m-1)}{(2^m)(2^m-1)} \\
&= \frac{1}{2^{m-1}}.
\end{aligned}$$

Therefore, using Theorem 2.6, we have:

$$S(n) = \left(\frac{2^{\min(e_i)\cdot m} - 1}{2^m - 1} + 1\right) \prod_{i=1}^{m} \gcd(d, d_i) \leq \frac{1}{2^{m-1}} \cdot 2^{\min(e_i)\cdot m} \prod_{i=1}^{m} \gcd(d, d_i) \quad (4.1)$$

$$\leq \frac{1}{2^{m-1}} \prod_{i=1}^{m} (2^{e_i} \cdot d_i) \quad (4.2)$$

$$= \frac{1}{2^{m-1}} \prod_{i=1}^{m} (p_i - 1)$$

$$\leq \frac{1}{2^{m-1}} \varphi(n). \quad (4.3)$$

We obtain equality in equation (4.1) above when $\min(e_i) = 1$ and in equation (4.2) when $e_1 = e_2 = \cdots = e_m$ and $\gcd(d, d_i) = d_i$ for all i (which is equivalent to $d_i \mid d$). We obtain equality in equation (4.3) when $\varphi(n) = \prod_{i=1}^{m}(p_i - 1)$. This occurs if and only n is square free. The result follows.

□

Remark 4.1. *For the case $m = 2$, Monier [108] remarked that the bound is met in this case for numbers of the form $n = (2x + 1)(4x + 1)$ with $2x + 1, 4x + 1$ prime and x odd, see also [113]. This form was exploited extensively in Chapter 3, but will be less useful in this chapter because we require numbers n of cryptographic size that satisfy a smaller smoothness bound. For example, we will be interested in constructing 1024-bit n in which each prime factor has at most 128 bits, meaning n will have at least 8 prime factors.*

We now go on to show that, when $m \geq 3$, the bound in the above theorem is attained if and only if n is a Carmichael number of special form. This result is of particular significance, as not only have we shown the maximum number of non-witnesses a generic odd composite number n with m distinct factors can have, we now provide a method to produce such a number that meets this bound for any m.

Theorem 4.4. *Let n be a Carmichael number with $m \geq 3$ prime factors that are all congruent to $3 \bmod 4$. Then $S(n) = \frac{\varphi(n)}{2^{m-1}}$. Conversely, if n has $m \geq 3$ prime factors and $S(n) = \frac{\varphi(n)}{2^{m-1}}$, then n is a Carmichael number whose prime factors are all congruent to $3 \bmod 4$.*

Proof. By Korselt's criterion we know that n is square-free. Write $n = p_1 \cdots p_m$

m	C_m	$S(C_m)$
3	$7 \cdot 19 \cdot 67$	$\varphi(C_m)/4$
4	$7 \cdot 19 \cdot 67 \cdot 199$	$\varphi(C_m)/8$
5	$7 \cdot 11 \cdot 19 \cdot 103 \cdot 9419$	$\varphi(C_m)/16$
6	$7 \cdot 11 \cdot 31 \cdot 47 \cdot 163 \cdot 223$	$\varphi(C_m)/32$
7	$19 \cdot 23 \cdot 31 \cdot 67 \cdot 71 \cdot 199 \cdot 271$	$\varphi(C_m)/64$
8	$11 \cdot 31 \cdot 43 \cdot 47 \cdot 71 \cdot 139 \cdot 239 \cdot 271$	$\varphi(C_m)/128$
9	$19 \cdot 31 \cdot 43 \cdot 67 \cdot 71 \cdot 103 \cdot 239 \cdot 307 \cdot 631$	$\varphi(C_m)/256$
10	$7 \cdot 11 \cdot 19 \cdot 31 \cdot 47 \cdot 79 \cdot 139 \cdot 163 \cdot 271 \cdot 2347$	$\varphi(C_m)/512$

Table 4.1: The smallest number C_m with m prime factors that meets the upper bound of $\varphi(C_m)/2^{m-1}$ on $S(C_m)$.

where the p_i are prime and, by assumption, $p_i = 3 \bmod 4$ for each i. As before, we write $n = 2^e d + 1$ where d is odd and $p_i = 2^{e_i} d_i + 1$ where each d_i is odd. Since $p_i = 3 \bmod 4$ for each i, it is immediate that $e_i = 1$ for each i. Moreover, by Korselt's criterion, we have $2^{e_i} d_i | 2^e d$, and hence $d_i | d$, for each i. The result follows from the converse part of Theorem 4.3.

For the converse, let $n = \prod_{i=1}^{m} p_i^{q_i}$. Suppose $p_i = 2^{e_i} d_i + 1$ where d_i is odd and $n = 2^e d + 1$ where d is odd. Necessarily, $e \geq 1$. By Theorem 4.3, since $S(n) = \frac{\varphi(n)}{2^{m-1}}$, we have that n is square free, $e_i = 1$ for all i and $d_i \mid d$ for all i. Since $e_i = 1 \; \forall i$, we have that $p_i = 3 \bmod 4$ and $2^{e_i} \mid 2^e$ for all i. Also, since $d_i \mid d$ for all i, it follows that $2_i^e d_i \mid 2^e d$ for all i, and thus $p_i - 1 \mid n - 1$ for all i. Hence, n satisfies Korselt's criterion, and n is a Carmichael number. □

Example 4.1. Table 4.1 gives, for each $3 \leq m \leq 10$, the smallest number with m prime factors achieving the bound of Theorem 4.3. In the light of Theorem 4.4, these are all Carmichael numbers whose prime factors are all congruent to 3 mod 4. These are obtained from data made available by Pinch and reported in [129]. Of course, these examples are all much too small for cryptographic use.

4.3 Generating Large Carmichael Numbers

The results in the previous section motivate the search for cryptographically-sized Carmichael numbers with a chosen number of prime factors, with each factor congruent to 3 mod 4. In this section, we discuss two existing constructions for Carmichael numbers: the Erdös method [51] and the method of Granville and Pomerance [67]. We show how to combine these two methods to produce large examples. We also show how to modify the constructions to improve the probability that they will succeed in constructing large examples by the introduction of sieving during the generation process.

4.3.1 The Erdös Method

Erdös [51] gave a method to construct Carmichael numbers with many prime factors. The method starts with a highly composite number L and then considers the set $\mathcal{P}(L) = \{p : p \text{ prime}, p-1 \mid L, p \nmid L\}$. If for some subset $\{p_1, p_2, \ldots, p_m\}$ of $\mathcal{P}(L)$, we have $p_1 p_2 \cdots p_m = 1 \bmod L$, then $n = p_1 p_2 \cdots p_m$ is a Carmichael number, by Korselt's criterion. This is easy to see: by construction, $p_i - 1 \mid L$; the condition $n = 1 \bmod L$ implies that $L \mid n-1$; it follows that $p_i - 1 \mid n - 1$, and n is evidently square-free.

Example 4.2. If $L = 120 = 2^3 \cdot 3 \cdot 5$, then $\mathcal{P}(L) = \{7, 11, 13, 31, 41, 61\}$. If we examine all the subsets of $\mathcal{P}(L)$, we find that $41040 = 7 \cdot 11 \cdot 13 \cdot 41$, $172081 = 7 \cdot 13 \cdot 31 \cdot 61$ and $852841 = 11 \cdot 31 \cdot 41 \cdot 61$ are all 1 mod 120, and so are all Carmichael numbers.

The Erdös method lends itself to a computational approach to generating Carmichael numbers with a chosen number of prime factors m for moderate values of L. For a given L, the set $\mathcal{P}(L)$ can be quickly generated by considering each factor f of the selected L and testing the primality of $f + 1$. One can then examine all m-products of distinct elements from $\mathcal{P}(L)$ and test the product n against the condition $n = 1 \bmod L$.

Alternatively, as pointed out in [26], one can employ a time-memory trade-off

(TMTO): for some k, build a table of all k-products $p_1 \cdots p_k$ from $\mathcal{P}(L)$, and look for collisions in that table with the inverses of $(m-k)$-products $(p_{k+1} \cdots p_m)^{-1} \bmod L$ from $\mathcal{P}(L)$. Such a collision gives an equation

$$p_1 \cdots p_k = (p_{k+1} \cdots p_m)^{-1} \bmod L$$

and hence

$$p_1 \cdots p_k p_{k+1} \cdots p_m = 1 \bmod L.$$

Of course, one needs to take care to avoid repeated primes in such an approach. For the L we use later, the direct approach suffices, and so we did not explore this direction further.

4.3.2 The Selection of L in the Erdös Method

Clearly, L must be even, otherwise the integers p satisfying $p-1 \mid L$ will all be even. We can ensure that all primes p in $\mathcal{P}(L)$ satisfy $p = 3 \bmod 4$ by setting the maximum power of 2 in L to be 1, i.e. by setting $L = 2 \bmod 4$. For then each factor f of L must be 2 mod 4, and hence $p = f + 1 = 3 \bmod 4$. As we shall see later, other conditions can be imposed on L as needed.

Note that since $2 \mid L$, $p = 3$ is a candidate for inclusion in $\mathcal{P}(L)$. However, if 3 is also a factor of L then it is excluded from $\mathcal{P}(L)$ because of the additional condition $p \nmid L$ on elements of $\mathcal{P}(L)$; this condition is needed in general, since if $p \mid L$, then any product $p_1 p_2 \cdots p_m$ including p as a factor would be 0 mod L instead of the required 1 mod L.

For the Erdös method to be successful in producing a Carmichael number with m prime factors, we need to find a product p_i such that $p_1 p_2 \cdots p_m = 1 \bmod L$. One can see that the number of possible products that can be considered is $\binom{|\mathcal{P}(L)|}{m}$. Let us make the heuristic assumption that the values of $p_1 p_2 \cdots p_m$ are uniformly distributed amongst the odd numbers modulo the even integer L. Then we need to ensure that:

$$\binom{|\mathcal{P}(L)|}{m} \gtrapprox L/2$$

for the method to have a reasonable chance of success, since we need some product equal to 1 mod L, and there are $L/2$ possible odd values mod L that a product may

L_{bound}	L_{best}	$\lvert\mathcal{P}(L_{best})\rvert$
2^{20}	$810810 = 2 \cdot 3^4 \cdot 5 \cdot 7 \cdot 11 \cdot 13$	39
2^{21}	$2088450 = 2 \cdot 3^3 \cdot 5^2 \cdot 7 \cdot 13 \cdot 17$	50
2^{22}	$4054050 = 2 \cdot 3^4 \cdot 5^2 \cdot 7 \cdot 11 \cdot 13$	58
2^{23}	$7657650 = 2 \cdot 3^2 \cdot 5^2 \cdot 7 \cdot 11 \cdot 13 \cdot 17$	65
2^{24}	$13783770 = 2 \cdot 3^4 \cdot 5 \cdot 7 \cdot 11 \cdot 13 \cdot 17$	73
2^{25}	$22972950 = 2 \cdot 3^3 \cdot 5^2 \cdot 7 \cdot 11 \cdot 13 \cdot 17$	89
2^{26}	$53603550 = 2 \cdot 3^2 \cdot 5^2 \cdot 7^2 \cdot 11 \cdot 13 \cdot 17$	93

Table 4.2: For a given L_{bound} (column 1), the value L_{best} (column 2) gives the value of $L \leq L_{bound}$ resulting in the largest set of primes $\mathcal{P}(L)$, subject to the additional restriction that $p = 3 \bmod 4$ for all $p \in \mathcal{P}(L)$.

take.

Thus it is desirable to find L such that $|\mathcal{P}(L)|$ is as large as possible. In turn, this heuristically depends on L being as smooth as possible, since such an L has many factors f and therefore many possible candidates $p = f + 1$ that, if prime, can be included in $\mathcal{P}(L)$. This analysis of course depends on the primality of the different values $f + 1$ being in some sense independent for the different factors f of L; this is a reasonable assumption given standard heuristics on the distribution of primes.

For various bounds L_{bound}, we have computed the value of $L \leq L_{bound}$ giving the largest set $\mathcal{P}(L)$, where we impose the restriction $L = 2 \bmod 4$ to ensure the primes in $\mathcal{P}(L)$ are all 3 mod 4. This was done by a brute-force search. The results are shown in Table 4.2, and bear out our heuristic analysis suggesting that smooth L make the best choices.

Example 4.3. Suppose $L = 53603550 = 2 \cdot 3^2 \cdot 5^2 \cdot 7^2 \cdot 11 \cdot 13 \cdot 17$. Then $|\mathcal{P}(L)| = 93$ with:

$$\begin{aligned}\mathcal{P}(L) = \{&19, 23, 31, 43, 67, 71, 79, 103, 127, 131, 151, 199, 211, 239, 307, 331, 443,\\ &463, 491, 547, 631, 859, 883, 911, 991, 1051, 1123, 1171, 1327, 1471, 1531,\\ &1667, 1871, 1951, 2003, 2143, 2311, 2551, 2731, 3571, 3823, 3851, 4951,\\ &4999, 5851, 6007, 7151, 7351, 8191, 9283, 10711, 11467, 11551, 16831,\\ &17851, 19891, 22051, 23563, 26951, 27847, 28051, 33151, 34651, 41651,\\ &42043, 43759, 46411, 50051, 53551, 54979, 57331, 72931, 77351, 91631\\ &102103, 117811, 124951, 126127, 150151, 232051, 242551, 286651,\\ &324871, 350351, 450451, 824671, 1051051, 1093951, 1191191, 1624351,\\ &2144143, 4873051, 10720711\}.\end{aligned}$$

As representative examples, the following Carmichael numbers with, respectively 8 and 16 prime factors, can then be obtained by running a simple search algorithm over subsets of $\mathcal{P}(L)$ to find subsets whose products are 1 mod L:

$$\begin{aligned}C_8 &= 19 \cdot 23 \cdot 43 \cdot 239 \cdot 859 \cdot 9283 \cdot 11467 \cdot 242551\\ C_{16} &= 19 \cdot 23 \cdot 31 \cdot 43 \cdot 67 \cdot 71 \cdot 79 \cdot 103 \cdot 127 \cdot 131 \cdot 491 \cdot 1531 \cdot 3851 \cdot 7151 \cdot 11467 \cdot 33151\end{aligned}$$

Here

$$C_8 = 9960524081137300040 3701$$

and

$$C_{16} = 29520757403834736752314039155478508748 01.$$

Our SAGE [148] implementation of the Erdös method running on a 3.3GHz processor took 4.83 seconds to find C_8 and 1.78 seconds to find C_{16}. The code used to generate these examples can be found in Appendix A.1.

It would be tempting to think that this method could easily be scaled-up to numbers of cryptographic size. However, this is not so easy. To illustrate, suppose we wanted to construct a 1024-bit n with, say, $m = 8$ prime factors, all having about 128 bits. This would necessitate using an L substantially larger than 2^{128}, which would make the direct approach of finding a product $p_1 \cdots p_8 = 1 \bmod L$ infeasible; even the TMTO version would require prohibitive time and memory, on the order of 2^{64} of each.

4.3.3 The Method of Granville and Pomerance

The second method of generating Carmichael numbers that we consider is due to Granville and Pomerance [67]. This takes a small Carmichael number with m (known) factors and produces from it a larger Carmichael number, also with m factors. It is based on the following theorem.

Theorem 4.5 (Granville and Pomerance [67])**.** *Let $n = p_1 p_2 \cdots p_m$ be a Carmichael Number. Let $L = \operatorname{lcm}(p_i - 1)$ and let M be any integer with $M \equiv 1 \bmod L$. Set $q_i = 1 + M(p_i - 1)$. Then $N = q_1 \cdots q_m$ is a Carmichael number whenever each q_i is prime.*

Recall that we are interested in Carmichael numbers N in which all prime factors are congruent to 3 mod 4. Fortunately, as the following lemma shows, the method of Granville and Pomerance 'preserves' this property.

Lemma 4.1. *With notation as in Theorem 4.5, suppose $p_i \equiv 3 \pmod 4$. Then $q_i = 3 \pmod 4$.*

Proof. The integer L is even as it is the least common multiple of even integers $p_i - 1$. But $M \equiv 1 \pmod L$ implies that M is odd; write $M = 2s + 1$. Moreover, since $p_i = 3 \bmod 4$, we have $p_i - 1 = 2d_i$ with d_i odd; write $d_i = 2t_i + 1$. Then $q_i = 1 + M(p_i - 1) = 1 + (2s + 1)(4t_i + 2) = 3 + 4(2st_i + s + t_i)$, which is evidently $3 \bmod 4$. □

There are two important choices of variable in this method: M and the starting Carmichael number n.

Clearly, the properties of the resulting Carmichael number N are dependent on n, for example the value of each prime factor mod 4 (as seen in Lemma 4.1) and the number m of these factors.

The effects of M are more subtle. In particular, we need to select an M such that all the resulting $q_i = 1 + M(p_i - 1)$ are prime. Using the heuristic that the values q_i are as likely to be prime as random choices of odd q_i of the same size, the probability

that a random choice of M yields m primes is approximately $(2/\ln(B))^m$ where B is a bound on the q_i. This probability drops very quickly for N of cryptographic size and even moderate m. For example, with B of 128 bits and $m = 8$ (so that the target N has 1024 bits), we obtain $(2/\ln(B))^m \approx 2^{-43.77}$. Clearly then, simply making random choices of M is unlikely to yield candidates of cryptographically interesting sizes in a reasonable amount of time. We therefore turn to investigating methods for improving the probability that the q_i are all prime by careful choice of M.

4.3.4 The Selection of M in the Method of Granville and Pomerance

The only requirement on M coming from Theorem 4.5 is that $M \equiv 1 \pmod{L}$, where $L = \text{lcm}(p_i - 1)$. However, by a careful choice of M we can both ensure that this is true, and that the resulting values $q_i = 1 + M(p_i - 1)$ are more likely to be prime than if M was chosen at random.

Our approach is inspired by techniques originally introduced in [81, 80] for generating primes on low-end processors. There, one considers numbers of the form $p = kH + \delta$ where H is smooth (say, H is the product of the first h primes, $H = \prod_{i=1}^{h} s_i$), δ is chosen to be co-prime to H, and k is a free parameter. Then p is guaranteed to be divisible by each of $s_1, \ldots, s_h$, since $p = \delta \neq 0 \bmod s_i$. By choosing different values of k, one can generate different candidates for p, and test them for primality. Numbers p generated in this way have a higher probability of being prime than uniformly random candidates, since they are effectively guaranteed to pass trial divisions by each of the small primes dividing H. We refer to this process as 'sieving' by the primes $s_1, s_2, \ldots, s_h$. An analysis using the inclusion-exclusion principle can be used to evaluate the increase in probability that can be achieved by this means; a factor of 5 increase is typical even for moderate values of h, since many small divisors can be eliminated.

We present an adaptation of this method to generate candidates for M in the method of Granville and Pomerance, such that the resulting q_i are guaranteed to be indivisible by many small primes.

Since $M = 1 \pmod{L}$, we can write $M = kL + 1$, where k now becomes the free parameter in the construction method. Then

$$q_i - 1 = M(p_i - 1) = (kL + 1)(p_i - 1) = kLp_i + p_i - kL - 1.$$

Rearranging, we get:

$$q_i = kLp_i + p_i - kL = kL(p_i - 1) + p_i.$$

Note that, typically, many small primes will divide L because L is the least common multiple of the $p_i - 1$. This is especially so if we use the Erdös method to generate the starting Carmichael number n, since it starts with a smooth number which all the $p_i - 1$ will divide.

Now none of the primes dividing L can be a p_i (again, because L is the least common multiple of the $p_i - 1$). For each such prime p, we have:

$$q_i = p_i \neq 0 \bmod p.$$

Hence, we are assured that q_i is not divisible by any of the prime divisors of L: we achieve 'free' sieving on q_i for every such divisor.

Now we consider other primes (not equal to any of the p_i, and not dividing L). Let s denote such a prime, and suppose we choose k such that s divides k. Recalling that $M = kL + 1$, then we get:

$$q_i = kL(p_i - 1) + p_i = p_i \neq 0 \bmod s.$$

Hence, by choosing k so that it is divisible by a product of primes s_j that do not equal any of the p_i nor any of the divisors of L, we also obtain sieving on all the s_j. Of course, we can include an additional factor when building k to ensure that the resulting q_i are of any desired bit-size and that there are sufficiently many choices for k (and thence M). In what follows, we write $k = k' \prod_j s_j$ for some collection of primes s_j subject to the above constraints; k' now replaces k as the free parameter in the construction.

The overall sieving effectiveness will be determined by the collection of prime factors present in L and the s_j. Let us denote the complete set of primes from these two sources as $\{s_1, \ldots, s_h\}$. Then the fraction of non-prime candidates for each q_i that are removed by the sieving can be calculated using the formula:

$$\sigma = 1 - \prod_{i=1}^{h} \left(1 - \frac{1}{s_i}\right). \tag{4.4}$$

This follows easily by noting that a fraction $1 - \frac{1}{s_i}$ of integers are not divisible by s_i, so the probability that a randomly sampled integer is *not* divisible by any of the s_i is $\prod_{i=1}^{h} \left(1 - \frac{1}{s_i}\right)$, and hence the probability that a randomly sampled odd integer is divisible by at least one s_i is σ. This means that the prime values of q_i are now concentrated in a fraction $1 - \sigma$ of the initial set of candidates, so that a random selection from this reduced set is $1/(1 - \sigma)$ times more likely to result in a prime. Notice that the effect here is multiplicative across all m of the q_i – they all benefit from the sieving on the s_i. Note too how powerful the prime $s = 3$ is in sieving, contributing a factor $2/3$ to the product term determining σ.

The overall effect is to improve the success probability for each trial of the modified Granville-Pomerance construction (involving a choice of k') from $(2/\ln(B))^m$ to $(2/(1 - \sigma)\ln(B))^m$.

Example 4.4. Using a C implementation of the modified Granville-Pomerance construction, with the Carmichael number C_8 of Example 4.3 as the starting value n and $L = 53603550$, we found that choosing

$$k = 7891867750444302551322686487$$

produces the 8-factor, 1024-bit Carmichael number $N = q_1 \cdots q_8$ where:

$$q_1 = 76145782959779164924491574423241193 19$$
$$q_2 = 93067068061952312685489702072850347 23$$
$$q_3 = 17767349357281805149048034032089611743$$
$$q_4 = 10068164635793022917793885951517446 6539$$
$$q_5 = 36296156544161401947340983808411635 4159$$
$$q_6 = 39265842079592789379396155210918041 94983$$
$$q_7 = 48504863745379328056901132907604640 05567$$
$$q_8 = 10260644253830242473575239653531750 7810051.$$

Here, q_8, the largest prime factor, has 137 bits.

As pointed out in Section 4.3.3, with B of 128 bits and $m = 8$ (so that the target N has 1024 bits), we estimate the standard Granville-Pomerance construction to have a success rate of $(2/\ln(B))^m \approx 2^{-43.8}$ per trial, so that the expected number of trials would be about $2^{43.8}$. With our modified version of the Granville-Pomerance construction we obtain sieving on each of the q_i by the primes $3, 5, 7, 11, 13, 17$ that divide L (in this case, we did not add any more primes to k to improve the sieving further). This gives us $\sigma = 0.6393$ and therefore reduces the expected number of trials by a factor of about $1/(1-\sigma)^m \approx 2^{11.8}$ to roughly 2^{32} trials. Finding the above N using our 'C' implementation actually took $2^{31.51}$ trials and less than one core-hour running on 3.3GHz CPUs.

The above example illustrates that we can generate numbers that are of cryptographically interesting size, have a controlled number of prime factors (and therefore achieve a given smoothness bound), achieve the upper bound of Theorem 4.3 on the number of Miller-Rabin non-witnesses, and hence maximise the probability of passing random-base Miller-Rabin primality tests.

4.4 Fooling Diffie-Hellman Parameter Validation in the Safe-Prime Setting

In this section, we target the problem of producing Diffie-Hellman parameters for the prime order setting, where the parameters are able to pass validity tests on the parameters but where the relevant Discrete Logarithm Problem (DLP) is relatively easy.

A Diffie-Hellman (DH) parameter set (p, q, g) in the prime order setting is formed of a prime p with $g \in \mathbb{Z}_p$ generating a group of prime order q, where $q \mid p - 1$. As explained in both Section 3.5 of Chapter 3 and Section 4.1 of this chapter, validating the correctness of DH parameters is vital in ensuring the subsequent security of the DH key exchange. As also explained there, Bleichenbacher [26] provided an extreme example of this in the context of Password Authenticated Key Exchange (PAKE): he showed that a client that accepts bad DH parameters in the SRP protocol can be subject to an offline dictionary attack on its password. Here, the attacker impersonates the server in a run of the SRP protocol, and induces the client to send a password-dependent protocol message; the attacker's ability to solve the DLP is what enables the offline password recovery.

DH validation checks should consist of primality tests on both p and q as well as a verification that $p = kq + 1$ for some integer k. The checks should also ensure that the given generator g generates the subgroup of order q. The security is based in part on size of q: it must still be large enough to thwart the Pohlig-Hellman algorithm for solving the DLP. For prime q, this algorithm runs in time $O(\sqrt{q})$.

In Chapter 3 we already showed how to subvert DH parameters in the case where k is permitted to be large and where a weak primality test based on Miller-Rabin with a small number of rounds is permitted. For example, we selected q to be of the form $(2x + 1)(4x + 1)$ with both factors prime, and then tried k of a suitable size until $kq + 1$ was prime. This gives an $O(q^{1/4})$ algorithm using the Pohlig-Hellman algorithm in the subgroups of orders $2x + 1$ and $4x + 1$, with q passing t rounds of random-base Miller-Rabin testing with the best possible probability 4^{-t} (this coming from the Monier-Rabin bound).

However, many implementations insist on using DH parameters in which p is a safe prime; that is, they require $p = 2q+1$, in which case g must have order q or $2q$ if it is not equal to ± 1. OpenSSL in its default setting is a good example of such a library. Insisting on safe primes to a large extent eliminates small subgroup attacks. It is also a good option in the context of protocols like SSL/TLS in which a server following the specification only provides p and g but not q.[2] As noted in the introduction of this chapter, the techniques of the previous chapter do not extend to the safe-prime setting, since they need the flexibility in k to force $p = kq + 1$ to be prime. The resulting q would also be too large and have too few prime factors to make the Pohlig-Hellman algorithm effective.

This leaves open the problem of fooling DH parameter validation upon safe prime parameter sets, when random-base Miller-Rabin tests are used for checking p and q (as should be the case in practice, in light of the work of [8] and [26]).

4.4.1 Generating Carmichael Numbers q such that $p = 2q + 1$ is Prime

To summarise the above discussion, we wish to construct a number q such that q and $p = 2q+1$ both pass random-base Miller-Rabin primality testing, and such that q is sufficiently smooth that the Pohlig-Hellman algorithm can be used to solve the DLP in some subgroup mod p.

Our approach parallels that of [26]: we construct q as a large Carmichael number with m prime factors that are all 3 mod 4 using the techniques from the previous section. Then q will pass random-base Miller-Rabin primality tests with the highest possible probability amongst all integers with m prime factors. After constructing a candidate q, we test $2q + 1$ for primality (using a robust primality test), rejecting q if this test fails, and stopping if it passes. If $2q + 1$ is prime, then the DLP in the subgroup of order q can be solved with $O(mB^{1/2})$ effort where B is an upper bound on the prime factors of q.

The approach just described will fail in practice. The first reason is that it is unlikely

[2] For if p is not a safe prime, then the client is forced to blindly accept the parameters or to do an expensive computation to factorise $p - 1$ and then test g for different possible orders arising as factors of $p - 1$. We know of no cryptographic library that does the latter.

that $2q+1$ will happen to be prime by chance (the probability is about $1/\ln q$ by standard density estimates for primes). The second reason is that there may be arithmetic reasons why $2q+1$ can *never* be prime. We investigate and resolve these issues next.

4.4.1.1 Sieving for $2q+1$

We begin by examining the method of Granville and Pomerance and its consequences for the values of $2q+1$ modulo small primes.

Assume we have some starting Carmichael number $n = p_1 \cdots p_m$, and we apply the method of Granville and Pomerance, setting $q_i = M(p_i - 1) + 1$ where $M = 1 + kL$ and $L = \operatorname{lcm}(p_i - 1)$. We assume k is such that the q_i are all prime, and we write $q = q_1 \cdots q_m$ for the resulting Carmichael number.

Lemma 4.2. *With notation as above, for all primes s dividing kL, we have that $2q + 1 = 2n + 1 \pmod{s}$.*

Proof. Since $q_i = M(p_i - 1) + 1 = (1 + kL)(p_i - 1) + 1$, it follows that for any prime s with $s \mid kL$ we have $q_i = p_i \pmod{s}$, therefore $2q + 1 \equiv 2n + 1 \pmod{s}$. □

The importance of the above lemma is that we can determine at the outset, based only on the small starting Carmichael number n, whether $2q + 1$ will be divisible by each of the primes s or not. In particular, we should just ignore any n for which $2n + 1 \equiv 0 \pmod{s}$ for any of the primes s dividing L or k, since then $2q + 1$ can never be prime. Typically, there are many such primes s, since L is usually rather smooth, arising as the least common multiple of the $p_i - 1$. This is particularly so when the Erdös method is used to construct n.

4.4.1.2 The Prime 3

The prime 3 plays a particularly important role when applying our sieving trick in the method of Granville and Pomerance: it contributes a factor $2/3$ to the product

term $\prod_{i=1}^{h}\left(1-\frac{1}{s_i}\right)$ when computing σ. It is therefore desirable to keep 3 as a factor of kL in the construction. On the other hand, the above lemma then imposes the necessary condition $2n+1 \neq 0 \bmod 3$ for $2q+1$ to be prime; this in turn requires $n = 0 \bmod 3$ or $n = 2 \bmod 3$.

We consider the two cases $n = 0 \bmod 3$ and $n = 2 \bmod 3$.

The case $n = 0 \bmod 3$: In this case, we have $3 \mid n$, and so we can set $p_1 = 3$. Recall that, in our approach, $n = p_1 \cdots p_m$ will be obtained using the Erdös method, in which case $p_1 = 3$ is contained in the set $\mathcal{P}(L^*)$ (henceforth L^* denotes the smooth number used in the Erdös method; we use L^* to distinguish it from $L = \mathrm{lcm}(p_i - 1)$ in the method of Granville and Pomerance – they are often equal but need not be so). From the conditions on $\mathcal{P}(L^*)$, we deduce that $3 \nmid L^*$. Since each prime in $\mathcal{P}(L^*)$ is constructed by adding 1 to a factor of L^*, we deduce that $p = 2 \bmod 3$ for every $p \in \mathcal{P}(L^*) \setminus \{3\}$. Since we will also have $p = 3 \bmod 4$ by choice of L^*, we deduce that $p = 11 \bmod 12$ for every $p \in \mathcal{P}(L^*) \setminus \{3\}$.

Hence, in the case where 3 appears as a factor in the starting Carmichael number n, and n is obtained via the Erdös method, then the remaining primes arising as factors of n must all be 11 mod 12. This happens automatically in the Erdös method simply by ensuring $3 \nmid L^*$.

The case $n = 2 \bmod 3$: In this case, we can show that $p_i = 2 \bmod 3$ for all primes p_i arising as factors of n. For suppose that $p_i = 1 \bmod 3$ for some i. This implies $3 \mid p_i - 1$. By Korselt's criterion, we deduce that $3 \mid n - 1$, and hence $n = 1 \bmod 3$. This contradicts our starting assumption on n.

Moreover, it is easy to see that we must take m, the number of prime factors of n, to be odd in this case. For $n = \prod_{i=1}^{m} p_i = 2^m \bmod 3$, and so $n = 2 \bmod 3$ if and only if m is odd.

Hence, in the case where $n = 2 \bmod 3$, we are forced to use a starting Carmichael number with m odd in which $p_i = 2 \bmod 3$ for each prime factor p_i (whether or not we use the Erdös method). This may sound overly restrictive. But, fortunately,

we have already seen how to arrange this for the Erdös method: we simply need to ensure that $3 \nmid L^*$, where L^* denotes the smooth number used in that construction, and then all but one of the primes $p \in \mathcal{P}(L^*)$ will satisfy this requirement. We then remove $p = 3$ from $\mathcal{P}(L^*)$ when running the last step in the Erdös method.

4.4.1.3 Other Primes

Of course, Lemma 4.2 imposes a single condition on n for every other prime s dividing kL, but these conditions are much less restrictive than that in the case $s = 3$, and so we do not investigate the implications for the p_i any further here.

4.4.1.4 Completing the Construction

We have now assembled all the tools necessary to produce a suitable Carmichael number n such that when the method of Granville and Pomerance is applied to produce q from n, then $2q + 1 \neq 0 \bmod 3$; moreover q will attain the bound of Theorem 4.3 on $S(q)$, the number of Miller-Rabin non-witnesses for q, namely $S(q) = \varphi(q)/2^{m-1}$. Our procedure is as follows:

1. We use the first step of the Erdös method with an L^* such that $2 \mid L^*$, $4 \nmid L^*$, $3 \nmid L^*$. This ensures that the resulting set $\mathcal{P}(L^*)$ contains the prime 3, and a collection of other primes that are all 11 mod 12.[3]

2. We remove 3 from $\mathcal{P}(L^*)$ and run the second step of the Erdös method with an odd m to find a subset of primes $p_1, \ldots, p_m$ such that $n := p_1 \cdots p_m = 1 \pmod{L}$; n is then a Carmichael number with m prime factors that are all 11 mod 12 and therefore both 3 mod 4 and 2 mod 3.

3. We set $L = \operatorname{lcm}(p_i - 1)$ and test the condition $2n + 1 \neq 0 \bmod s$ for each prime factor s of L (cf. Lemma 4.2). If any test fails, we go back to the previous step and generate another n.

[3] Of course, one could choose not to restrict L^* in this way and just filter the resulting set $\mathcal{P}(L^*)$ for primes that are 11 mod 12, but this involves wasted computation and the use of larger L^* than is necessary.

4. Integer n is then used in the method of Granville and Pomerance to produce candidates for q (in which the q_i are all prime). By construction of the p_i, we will have $3 \nmid L$ in the Granville-Pomerance method, but we desire $3 \mid kL$ in view of the power of sieving by 3 in that method. We therefore set $k = 3k'$ for k' of suitable size when running this step, introducing the prime 3 in k.

5. Finally, we test $2q + 1$ for primality. By choice of n, we are guaranteed that $2q + 1 \neq 0 \bmod 3$ and $2q + 1 \neq 0 \bmod s$ for each prime divisor s of L, so we are assured that $2q + 1$ will not be divisible by certain (small) primes.

Note that the procedure as described focusses on the case $n = 2 \bmod 3$. An alternative procedure could be developed for the case $n = 0 \bmod 3$. The procedure can be enhanced by setting k at step 4 to contain additional prime factors s beyond 3 not already found in L, to increase the effect of sieving. Of course, in view of Lemma 4.2, certain bad choices of s should be avoided at this stage.

4.4.2 Examples of Cryptographic Size

Using the method described above, we now give two examples of Carmichael numbers q such that $p = 2q + 1$ is a 1024-bit prime. In the first example q is the product of 9 prime factors, which by construction will pass a random-base Miller-Rabin primality test with probability approximately $1/2^8$. Since the largest factor of q is 121 bits in size, the DLP in the subgroup of order q mod p for this parameter set can be solved in approximately $9 \cdot 2^{60.5} \approx 2^{64}$ operations. In the second example, q is the product of 11 prime factors, which by construction will pass a random-base Miller-Rabin primality test with probability approximately $1/2^{10}$. However, because the q with 11 factors is smoother, with largest factor 100 bits in size, the DLP in the subgroup of order q mod p for this parameter set can be solved in approximately $11 \cdot 2^{50} \approx 2^{54}$ operations. We give both these examples to illustrate the trade-off between the probability of a parameter set being accepted and the work required to solve the DLP for that parameter set.

Example 4.5. Using SAGE [148] we examined all $L^* < 2^{30}$ such that $2 \mid L^*$, $4 \nmid L^*$, $3 \nmid L^*$. We found the largest set of primes $\mathcal{P}(L^*)$ was produced when

$L = 565815250 = 2 \cdot 5^3 \cdot 7^2 \cdot 11 \cdot 13 \cdot 17 \cdot 19$. Here, $|\mathcal{P}(L^*)| = 53$ (including the prime 3).

Then, using the Erdös method with $L^* = 565815250$ we generated the 9-factor Carmichael number

$$\begin{aligned} n &= 1712969394960887942534921587572251 \\ &= 71 \cdot 131 \cdot 647 \cdot 1871 \cdot 4523 \cdot 4751 \cdot 46751 \cdot 350351 \cdot 432251. \end{aligned}$$

Using the procedure described above, we found that $k = 3k'$ with

$$k' = 1844674409176776955124$$

produced a 9-factor, 1023-bit Carmichael number q such that $n = 2q+1$ is a 1024-bit prime.

To generate a target q with 1023 bits, with $m = 9$ factors each around 114 bits in size, we estimate the standard Granville-Pomerance construction to have a success rate of $(2/\ln(B))^m \approx 2^{-47.73}$ per trial, so that the expected number of trials would be about $2^{47.7}$. With our modified version of the Granville-Pomerance construction we obtain sieving on each of the q_i by the primes $5, 7, 11, 13, 17, 19$ that divide L and the prime 3 since it divides k. This gives us $\sigma = 0.658$ and therefore reduces the expected number of trials by about $1/(1-\sigma)^m \approx 2^{13.9}$ to roughly $2^{33.8}$ trials. We then need to consider the probability that the q produced is such that $p = 2q + 1$ is also prime. By Lemma 4.2 we know that we obtain sieving on $2q+1$ from all primes $s \mid kL$, hence a success rate of $(2/(1-\sigma)\ln(2^{1024})) \approx 2^{-6.9}$. Therefore we expect to require $2^{33.8+6.9} = 2^{40.7}$ total trials. Finding the above q such that $p = 2q + 1$ is prime actually took $2^{38.15}$ trials, so we were somewhat lucky. Our implementation is in 'C' and ran for 136 core-days on 3.2GHz CPUs.

The factors of this q are:

$$\begin{aligned}
q_1 &= 219186431519361672882122216610071\\
q_2 &= 407060515678814535352512687990131\\
q_3 &= 202277763945010915259787074185864 7\\
q_4 &= 585540895630294754699383635801 1871\\
q_5 &= 141594434761507640681850951930105 23\\
q_6 &= 148733649959566849455725789842547 51\\
q_7 &= 146385223907573688674845908950296751\\
q_8 &= 109702808975440517277502169413340 0351\\
q_9 &= 135347621463205833004710468756718 2251.
\end{aligned}$$

Since $2^q \equiv 1 \pmod{p}$ we can set a generator $g = 2$ to obtain a complete set of DH parameters (p, q, g). By construction q will pass a random-base Miller-Rabin primality test with probability approximately $1/2^8$. Since q_9, the largest factor of q, is 121 bits in size, the DLP in the subgroup of order q mod p for this parameter set can be solved in approximately $9 \cdot 2^{60.5} \approx 2^{64}$ operations.

The C code used to generate this example can be found in Appendix A.2.

Example 4.6. Again, using the Erdös method with $L^* = 565815250$ we generated the 11-factor Carmichael number

$$\begin{aligned}
n &= 96647594591145401276131753609264751\\
&= 23 \cdot 71 \cdot 191 \cdot 419 \cdot 491 \cdot 3851 \cdot 4523 \cdot 4751 \cdot 9311 \cdot 17291 \cdot 113051.
\end{aligned}$$

Using the procedure described above, we found that $k = 3k'$ with

$$k' = 3994916512074331$$

produced a 11-factor, 1023-bit Carmichael number q such that $p = 2q + 1$ is a 1024-bit prime.

To generate a target q with 1023 bits, with $m = 11$ factors each around 93 bits in size, we estimate the standard Granville-Pomerance construction to have a success rate of $(2/\ln(B))^m \approx 2^{-55.11}$ per trial, so that the expected number of trials would be about $2^{55.1}$. Again, using our modified version of the Granville-Pomerance construction

we sieve as in the previous example to reduce the expected number of trials by about $1/(1-0.658)^m \approx 2^{17}$ to roughly $2^{38.1}$ trials. Then again by considering the probability that the q produced is such that $2q+1$ is also prime we expect to require $2^{38.1+6.9} = 2^{45}$ total trials. Finding the above q such that $2q+1$ was prime took $2^{44.83}$ trials. The computation using our 'C' implementation ran for 1680 core-days on 3.3GHz CPUs.

The factors of this q are:

$$\begin{aligned}
q_1 &= 149185389210558730480951523 \\
q_2 &= 474680783851777778803027571 \\
q_3 &= 128841927045482539960821769 1 \\
q_4 &= 283452239500061587913807891 9 \\
q_5 &= 332276548696244445162119299 1 \\
q_6 &= 261074431118477778341665163 51 \\
q_7 &= 306643786368248445106755810 23 \\
q_8 &= 322104817613706349902054422 51 \\
q_9 &= 631325442522864445808026668 11 \\
q_{10} &= 117246153611389111364347809791 \\
q_{11} &= 766609465920621112766889525551.
\end{aligned}$$

Since $2^q \equiv 1 \pmod{p}$ we can set a generator $g = 2$ to obtain a complete set of DH parameters (p, q, g). By construction q will pass a random-base Miller-Rabin primality test with probability approximately $1/2^{10}$. Since q_{11}, the largest factor of q, is 100 bits in size, the DLP in the subgroup of order q mod p for this parameter set can be solved in approximately $11 \cdot 2^{50} \approx 2^{54}$ operations.

4.4.3 Application to OpenSSL and PAKE protocols

OpenSSL provides the DH parameter verification function `DH_check` in `dh_check.c`. This function takes a DH parameter set (p, q, g) and performs primality testing on both p and q. A safe-prime setting is enforced by default, and if q is not provided then it is calculated from p via $q = (p-1)/2$. For this reason, we were not able to create malicious DH parameter sets passing OpenSSL's testing using the approach

in Chapter 3. However, using the work of this chapter we are now able to create such parameter sets.

The primality test that OpenSSL uses is `BN_is_prime_ex`; this performs t rounds of random-base Miller-Rabin testing, where t is determined by the bit-size of p and q. Since p and q are 1024 and 1023 bits respectively, $t = 3$ rounds of Miller-Rabin are performed, at least in versions prior to OpenSSL 1.1.1pre9, 1.1.0i and 1.0.2p (released 14th August 2018). From versions 1.1.1pre9, 1.1.0i and 1.0.2p onwards, t was increased to 5, with the aim of achieving 128 bits of security instead of 80 bits.[4] This change was made based from disclosures of the findings of Chapter 3 to OpenSSL: the numbers 3 and 5 were selected based on estimates for the average case performance of Miller-Rabin primality testing, with the OpenSSL developers implicitly assuming that p and q are generated randomly rather than maliciously.

For the DH parameter set given in Example 4.5, we know that q has $\varphi(q)/2^8$ Miller-Rabin non-witnesses, and thus a probability of approximately $1/2^8$ of being declared prime by a single round of Miller-Rabin testing. Hence this DH parameter set will be accepted by `DH_check` as being valid with probability approximately 2^{-24} (and the lower probability of 2^{-40} in versions 1.1.1pre9, 1.1.0i and 1.0.2p of OpenSSL).

This may seem like a small probability, and indeed it is in a scenario where, say, malicious DH parameters are hard-coded into a server by a developer with the hope of later compromising honestly established TLS sessions between a client and a server: only 1 in 2^{24} sessions would be successfully established, and the malicious DH parameters would be quickly spotted if ever careful validation were to be carried out.

Consider instead a PAKE scenario like that envisaged by Bleichenbacher [26]. Here, a client and server use some hypothetical PAKE protocol which relies on DH parameters as part of the protocol, with the server supplying the DH parameters. Assume OpenSSL's DH parameter validation is used by the client. Then an attacker impersonating the server to the client has a 1 in 2^{24} chance of fooling the client into using a weak set of DH parameters. For specific PAKE protocols, this may allow the client's password to be recovered thereafter. For example, this is the case for

[4] Interestingly, the last time these iteration counts were changed was in February 2000 (OpenSSL version 0.9.5), before which they were all 2, independent of the bit-size of the number being tested.

SRP [162, 152], as seen in [26]. It is also true of J-PAKE [72]: in this protocol, the client in a first flow sends values $g_1 = g^{x_1}, g_2 = g^{x_2}$, while the server sends $g_3 = g^{x_3}, g_4 = g^{x_4}$ (along with proofs of knowledge of the exponents). In the second flow in J-PAKE, the client sends $(g_1 g_3 g_4)^{x_2 s}$ where s is the password or a derivative of it. At this point, the attacker aborts the protocol, and uses its ability to solve the DLP to recover x_2 from the first flow and then again to recover $x_2 s$ and thence s from the second flow.

We pick SRP and J-PAKE here only as illustrative examples; many other protocols would be similarly affected. We also note that the specification for using SRP in TLS [152] makes careful mention of the need to use trusted DH parameters, and gives examples of suitable parameter sets. However, [152] states that *clients SHOULD only accept group parameters that come from a trusted source*, leaving open the possibility for implementations to use parameters from untrusted sources (to remove that possibility the IETF reserved term "MUST" should have been used). Meanwhile J-PAKE [72] just assumes that the DH parameters are agreed in advance and suggests some methods and sources for obtaining parameters. This does not remove the possibility of the parties using bad parameters and side-steps the important problem of parameter verification.

The power of the attack in the PAKE scenario is that the client has a secret that an attacker would like to learn; the attacker then gains an advantage by impersonating the server in a standard attack scenario. This is different from a protocol like TLS where there is no such static secret and the server is usually authenticated and therefore hard to impersonate; there we require a "malicious developer" attack scenario.

The attack can be carried out repeatedly to boost its success probability, and it can be done across a large population of users in a stealthy manner. Thus even a small per-attempt success probability of 2^{-24} may represent a significant weakness in practice.

4.4.4 OpenSSL Disclosure and Mitigations

As remediation to attacks of this form, we recommend that OpenSSL and other cryptographic libraries modify their DH parameter testing code to carry out stronger primality tests – as our analysis shows, 3 rounds of random-base Miller-Rabin testing is insufficient; 5 rounds are better in that it reduces the success probability of our attack to 2^{-40}, but this is still far from the 128-bit security level that the OpenSSL developers have targeted.

As part of ongoing work with the developers of OpenSSL to improve security within primality testing and prime parameter validation, we disclosed the findings of this work to OpenSSL. This resulted in a contribution to the OpenSSL codebase by a pull request[5] to increase the number of rounds of Miller-Rabin performed during the primality test on Diffie-Hellman parameters p and q during the check found in `DH_check`. This update modified the primality tests within `DH_check` to perform at the 128-bit security level, by replacing the call to set the number of Miller-Rabin rounds on the bit-size of the parameters with an enforced 64 rounds. This request was accepted by reviewers and merged into OpenSSL in March 2019 and was utilised as part of OpenSSL 1.1.1c in May 2019. A preferable solution would have been to change the primality test itself to be safe under all use cases rather than to make a bespoke change to the DH parameter testing, and this is something we go on to address in Chapter 5.

[5] see `https://github.com/openssl/openssl/commit/2500c093aa1e9c90c11c415053c0a27a00661d0d`.

4.5 The Elliptic Curve Setting

An elliptic curve over a prime field $\mathbb{F}_p$ in short Weierstrass form is the set of solutions $(x, y) \in \mathbb{F}_p \times \mathbb{F}_p$ satisfying an equation of the type $y^2 = x^3 + ax + b$, where $a, b \in \mathbb{F}_p$ satisfy $4a^3 + 27b^2 \neq 0$, together with the point at infinity $\mathcal{O}$. When using a scheme such as Elliptic Curve Diffie-Hellman (ECDH), one typically transmits a description of the used curve via a set of domain parameters as part of the protocol, uses hard-coded parameters, or uses a standardised 'named' curve. An ECDH parameter set is typically composed of (p, E, P, q, h), where E is a description of the elliptic curve equation (typically represented by a and b), P is a base point that generates a subgroup of order q on the curve and h is the cofactor of this subgroup.

Analogously to our attacks on the parameter sets on finite field DH, we can create malicious ECDH parameter sets. The idea is to first construct a composite number q that is designed to be declared 'probably prime' by a target implementation of a probabilistic primality test but which is actually reasonably smooth, then retroactively construct a curve of suitable order $n = h \cdot q$. This can be done using the algorithm of Bröker and Stevenhagen [32].

Depending on the specific structure of n, a composite order will expose ECDH to attacks like Lim-Lee style small subgroup attacks as in [92], or may aid in solving the Elliptic Curve Discrete Logarithm Problem (ECDLP) in the order q subgroup. For this we would use the Pohlig-Hellman algorithm to solve ECDLP in time $O(B^{1/2})$ where B is an upper bound on the largest prime factor of q. For example, we could produce a 256-bit q with 4 prime factors, and hope to use the algorithm of Bröker and Stevenhagen to find a suitable curve over a 256-bit prime p of order $n = h \cdot q$ possibly even with $h = 1$. During parameter validation, q would pass a single round of the Miller-Rabin test with probability 1/8. And the ECDLP could be solved with effort approximately $4 \cdot 2^{32} = 2^{34}$ group operations.

4.5.1 The Algorithm of Bröker and Stevenhagen

This subsection was written by Steven Galbraith. Steven provided us with the methodology of creating elliptic curves in this way as well as the implementation included in

Appendix A.3. I provided Steven with the pseudoprime parameters that matched his requested criteria (e.g. a composite number n with the highest probability of passing a Miller-Rabin test, such that n was 256-bit and had 4 composite factors) which allowed Steven to produce the Examples in Section 4.5.2.

For completeness, we give a short exposition of the algorithm of Bröker and Stevenhagen [32].

An elliptic curve E over $\mathbb{F}_p$ has $\#E(\mathbb{F}_p) = p + 1 - t$ points where $|t| < 2\sqrt{p}$. The endomorphism ring of E contains $\mathbb{Z}[\sqrt{t^2 - 4p}]$, which is a subring of the imaginary quadratic field $K = \mathbb{Q}(\sqrt{t^2 - 4p})$. Conversely, if E is an elliptic curve over a number field whose endomorphism ring is the ring of integers of K, then (by the Complex Multiplication theory of elliptic curves) the reduction modulo p of E is an elliptic curve over $\mathbb{F}_p$ and, by taking a suitable isomorphism (a twist), we may ensure that the reduced curve has $p + 1 - t$ points.

The algorithm of Bröker and Stevenhagen exploits these ideas. Given an integer n, the first step is to construct a prime p and an integer t such that $p + 1 - t = n$ and such that $\mathbb{Q}(\sqrt{t^2 - 4p})$ has small discriminant D. Once this is done, the curve E is constructed using standard tools in Complex Multiplication (namely the Hilbert class polynomial).

We now briefly sketch the first step of the algorithm. The input is an integer n, and we wish to construct an elliptic curve with n points.

Let $D < 0$ be a discriminant of an imaginary quadratic field. We will try to find (p, t) such that $t^2 - 4p = f^2 D$ for some $f \in \mathbb{N}$. We also need $p + 1 - t = n$ and so $p = n + t - 1$. If $t^2 - 4p = f^2 D$ then

$$(t-2)^2 - f^2 D = t^2 - f^2 D - 4t + 4 = 4(p - t + 1) = 4n.$$

Hence, to construct a curve with n points it suffices to choose a discriminant D, solve the equation $w^2 - f^2 D = 4n$, and then check whether $n + (w + 2) - 1 = n + w + 1$ is prime. Note that if $\ell \mid n$ then $w^2 - f^2 D \equiv 0 \pmod{\ell}$ and so $(\frac{D}{\ell}) \neq -1$.

An important ingredient is Cornacchia's algorithm [37], which solves the equation

$w^2 - f^2 D = 4n$ (note that $D < 0$, so the left hand side is positive definite and the equation only has finitely many solutions). Cornacchia's algorithm starts by taking as input an integer x_0 such that $x_0^2 \equiv D \pmod{4n}$.

Putting everything together, the algorithm is as follows (we refer to [32] for the full details). Let $n = \ell_1 \cdots \ell_k$ be the target group order. Search over all $D < 0$ such that $D \equiv 0, 1 \pmod 4$, up to some bound $|D| < D_{\text{bound}}$. Ensure that $(\frac{D}{\ell_i}) \geq 0$ for all $\ell_i \mid n$. Determine all solutions $x_0 \in \mathbb{Z}/4n\mathbb{Z}$ such that $x_0^2 \equiv D \pmod{4n}$ and run Cornacchia's algorithm for each. Whenever we find an integer solution $w^2 - f^2 D = 4n$ check whether $p = n + w + 1$ is prime. If so, output (p, t).

Note that the algorithm is not guaranteed to succeed for a given integer n, because we are restricting to $|D| < D_{\text{bound}}$. In our application this is not a serious problem, because we are able to generate many viable choices for n.

In practice one usually desires elliptic curves of order q (supposed to be prime) or whose group order is $4q$ (Edwards and Montgomery curves have group order divisible by 4). We make one remark about the case when $n = 4q$ is even. If D is odd then any solution (w, f) to $w^2 - f^2 D = 4n$ has w odd, and so t is odd. If n is odd then this means $p = n + w + 1$ is odd, which is all good, whereas if n is even then p cannot be prime when D is odd, so when n is odd we must use odd discriminants D. On the other hand, when n is even then we can take D even (so that w and t will be even and so $p = n + w + 1$ will be odd).

4.5.2 Examples

We implemented the algorithm of Bröker and Stevenhagen [32] in SAGE, and ran it with q that are 256-bit Carmichael numbers with 3 and 4 prime factors, all congruent to 3 mod 4. These were generated using methods described in Section 4.3. By design, these values of q pass random-base Miller-Rabin primality testing with probability 1/4 and 1/8 per iteration, respectively. We used an early abort strategy for each q and estimate a success probability of roughly 1/4 for each q we tried. When successful, the computations took less than a minute on a laptop. The SAGE code for the first stage (finding p, t) of the 3-prime case can be found in the Appendix A.3.

Example 4.7. Set $q = q_1 q_2 q_3$ where:

$$q_1 = 120969320416809549586937 71$$
$$q_2 = 362907961250428648760813 11$$
$$q_3 = 133066252458490504545631471$$

Then q is a Carmichael number with 3 prime factors that are all congruent to 3 mod 4, so q passes random-base Miller-Rabin primality testing with probability 1/4 per iteration. Using the algorithm of Bröker and Stevenhagen, we obtain the elliptic curve $E(\mathbb{F}_p)$ defined by $y^2 = x^3 + 5$, where

$$p = 584170554761513436280134435700062590071846222494668956566359474640363466559 53$$

such that $\#E(\mathbb{F}_p) = q$ and p has 256 bits. Every point P on this curve satisfies $[q]P = \mathcal{O}$, the point at infinity, so any point can be used as a generator (of course such points may not have order q, but if q is accepted as being prime then this will not matter). The Pohlig-Hellman algorithm can be used to solve the ECDLP on this curve using about $3 \cdot 2^{42.5}$ group operations, since the largest prime factor of q has 85 bits.

Example 4.8. Set $q = q_1 q_2 q_3 q_4$ where:

$$q_1 = 2758736250382478263$$
$$q_2 = 8276208751147434787$$
$$q_3 = 30346098754207260883$$
$$q_4 = 91038296262621782647$$

Then q is a Carmichael number with 4 prime factors that are all congruent to 3 mod 4, so q passes random-base Miller-Rabin primality testing with probability 1/8 per iteration. Using the algorithm of Bröker and Stevenhagen, we obtain the elliptic curve $E(\mathbb{F}_p)$ defined by $y^2 = x^3 + 2$, where

$$p = 630766480273645340284659517403254049576129731687884275351051601579812429521 39$$

such that $q = \#E(\mathbb{F}_p)$ and p has 256 bits. Every point P on this curve satisfies $[q]P = \mathcal{O}$, the point at infinity, so any point can be used as a generator. The Pohlig-

Hellman algorithm can be used to solve the ECDLP on this curve using about $4 \cdot 2^{33.5}$ group operations, since the largest prime factor of q has 67 bits.

The two examples above both construct examples of order q. We were also able to construct examples of order $4q$, compatible with applications that use Montgomery or Edwards curves, see for example [21, 28].

Example 4.9. Set $q = q_1 q_2 q_3 q_4$ where:

$$q_1 = 2758736250261979423$$
$$q_2 = 8276208750785938267$$
$$q_3 = 30346098752881773643$$
$$q_4 = 91038296258645320927$$

Then q is a Carmichael number with 4 prime factors that are all congruent to 3 mod 4, so q passes random-base Miller-Rabin primality testing with probability 1/8 per iteration. Using the algorithm of Bröker and Stevenhagen, we obtain the elliptic curve $E(\mathbb{F}_p)$ defined by:

$$\begin{aligned} y^2 = x^3 \\ &+ 6321182879949803182120490422518156174809202662482030327699487279440770594341 8x \\ &+ 2497861913915599596071303634889932909265875530926177668881756252856689679445 16 \end{aligned}$$

where

$$p = 2523065920653761378186867007329666643250876945351766405414005544192870317344 61,$$

such that $4q = \#E(\mathbb{F}_p)$ and p has 256 bits. Every point P on this curve satisfies $[4q]P = \mathcal{O}$, the point at infinity, so any point can be used as a generator. The Pohlig-Hellman algorithm can be used to solve the ECDLP on this curve using about $4 \cdot 2^{33.5}$ group operations, since the largest prime factor of q has 67 bits.

We have not attempted to do it, but we see no reason why similar examples could not be constructed where q passes fixed-base Miller-Rabin primality tests with probability 1, as per [26].

These examples illustrate the necessity for careful parameter validation, in particular robust primality testing of q, when accepting bespoke curves in cryptographic applications.

4.6 Conclusion and Recommendations

The best countermeasure to malicious DH and ECDH parameter sets is for protocols and systems to use only widely vetted sets of parameters, and to eliminate any options for using bespoke parameters. This is already widely done in the elliptic curve setting, not necessarily because parameter validation is hard, but because suitable parameter generation is non-trivial in the first place, and because safe and efficient implementation is much easier with a limited and well-understood set of curves. Nevertheless, issues can still arise with the provenance of parameter sets. In short, it is difficult to eliminate suspicion that a curve may have a hidden backdoor unless the generation process is fully explained and has demonstrably little opportunity for manipulation; see [20] for an extensive treatment. Similar concerns apply in the finite field setting, in the light of [63, 54].

On the flip-side is the argument that, in the finite field setting, using a common set of DH parameters may be inadvisable because, with the best known algorithms for finding discrete logarithms, the cost of solving many logarithms can be amortised over the cost of a large pre-computation, making commonly used DH parameter an even more attractive target. This was a crucial factor in assessing the impact of the Logjam attack on 512-bit DH arising in export cipher suites in TLS [3].

Our work adds to the weight of argument in favour of using only limited sets of carefully vetted DH parameters even in the finite field setting. This approach was recently adopted in TLS 1.3, for example, which in contrast to earlier versions of the protocol only supports a small set of DH and ECDH parameter sets, with the allowed DH parameters being specified in [61].

If bespoke parameters must be used, then implementations should employ robust primality testing as part of parameter validation, using, for example, at least 64 rounds of Miller-Rabin tests, or the Baillie-PSW primality test for which there are no known pseudoprimes, as discussed in Chapter 3.

Chapter 5

Sense and Securability: A Performant, Misuse-Resistant API for Primality Testing

Contents

In this chapter we set out to design a performant primality test that provides strong security guarantees across all use cases and that has the simplest possible API. We examine different options for the core of our test, describing four different candidate primality tests and analysing them theoretically and experimentally. We then evaluate the performance of the chosen test in the use case of prime generation and discuss how our proposed test was fully adopted by the developers of OpenSSL through a new API and primality test scheduled for release in OpenSSL 3.0 (2020).

5.1 Introduction and Motivation

Primality testing, and closely related tasks like random prime generation and testing of Diffie-Hellman parameters, are core cryptographic tasks. Primality testing is by now very well understood mathematically; there is a clear distinction between

accuracy and running time of different tests in settings that are malicious (i.e. where the input may be adversarially-selected) and non-malicious (e.g. where the input is random, as is common in prime generation).

Yet the results of Chapter 3 on how primality testing is actually done in practice have highlighted the failure of popular cryptographic libraries to provide primality testing APIs that are "misuse-resistant", that is, which provide reliable results in all use cases even when the developer is crypto-naive. Extending Chapter 3, in Chapter 4 we showed how failure to perform robust primality testing in the popular OpenSSL library has serious security consequences in the face of maliciously generated Diffie-Hellman parameter sets (see also Bleichenbacher [26] for an earlier example involving the GNU Crypto library).

The main underlying issue identified in Chapter 3 is that, while all libraries examined performed well on random inputs, some failed miserably on maliciously crafted ones in their default settings. Meanwhile code documentation was generally poor and did not distinguish clearly between the different use cases. And developers were faced with complex APIs requiring them to understand the distinctions between use cases and choose parameters to the APIs accordingly. An illustrative example is provided by the OpenSSL 1.1.1c primality testing code. This requires the developer using the function `BN_is_prime_fasttest_ex`[1] to pass multiple parameters, including `checks`, the number of rounds of Miller-Rabin testing to be carried out; and `do_trial_division`, a flag indicating whether or not trial division should be performed. Setting `checks` to 0 makes the test default to using a number of rounds that depends only on the size of the number being tested;[2] then the number of rounds *decreases* as the size increases, this being motivated by average-case error estimates for the Miller-Rabin primality test operating on random numbers (see Table 3.4 in Section 3.3.1). This makes the default setting performant for random prime generation, but dangerous in potentially hostile settings, e.g. Diffie-Hellman parameter testing.

As an illustration of how this can go wrong in practice, in Section 4.4.3 we pointed

[1] See `https://github.com/openssl/openssl/blob/3e3dcf9ab8a2fc0214502dad56d94fd95bcbbfd5/crypto/bn/bn_prime.c#L186`.

[2] Strictly, the default is invoked by setting `checks` to `BN_prime_checks`, an environmental variable that is set to 0.

out that OpenSSL (pre-1.1.1c May 2019) itself makes the wrong choice in using the default setting when testing finite field Diffie-Hellman parameters. We then went on and exploited this choice to construct Diffie-Hellman parameter sets (p, q, g) of cryptographic size that fool OpenSSL's parameter validation with a non-trivial success rate. In response to our work, the Diffie-Hellman parameter validation in OpenSSL 1.1.1c was subsequently changed to remedy this issue (though without changing the underlying primality test).[3] This example provides *prima facie* evidence that even very experienced developers can misunderstand how to correctly use complex primality testing APIs.

One may argue that developers who are not cryptography experts should not be using such security-sensitive APIs. However, they inevitably will, and, as our OpenSSL example shows, even expert developers can get it wrong. This motivates the search for APIs that are "misuse-resistant" or "robust", and that do not sacrifice performance (too much). This search accords with a long line of work that identifies the problem of API design as being critical for making it possible for developers to write secure cryptographic software (see [70, 164, 68] amongst others).

5.1.1 Contributions

Given this background, we set out to design a performant primality test that provides strong security guarantees across all use cases and that has the simplest possible API: it takes just one input, the number being tested for primality, and returns just one integer (or Boolean) indicating that the tested number is highly likely to be prime (1) or is definitely composite (0). We note that none of the many crypto libraries examined in Chapter 3 provide such an API.

We examine different options for the core of our test – whether to use many rounds of Miller-Rabin (MR) testing (up to 64 or 128, to achieve false positive rates of 2^{-128} or 2^{-256}, respectively), or to rely on a more complex primality test, such as the Baillie-PSW test [134] which combines MR testing with a Lucas test. Based on a combination of code simplicity, performance and guaranteed security, we opt for 64 rounds of MR as the core of our test.

[3] See `https://github.com/openssl/openssl/pull/8593` for our contribution to OpenSSL on this issue.

We also study the performance impact of doing trial division prior to more expensive testing. This is common practice in primality testing code, with the idea being that one can trade fast but inaccurate trial division for much slower but more accurate number theoretic tests such as Miller-Rabin. For example, OpenSSL 1.1.1c tests for divisibility using a fixed list of the first 2047 odd primes. We show that this is a sub-optimal choice when testing random inputs of common cryptographic sizes, and that the running time can be reduced substantially by doing trial division with fewer primes. The optimal amount of trial division to use depends on the size of the input being tested, though is not a new observation – see for example [105, 102, 82]. What is more surprising is that OpenSSL chooses so conservatively and with a fixed list of primes (independent of the input size). For example, with 1024-bit random, odd inputs, trial division using the first 128 odd primes already removes about 83% of candidates, while extending the list to 2047 primes, as OpenSSL does, only removes a further 5.5%. On average, it turns out to be faster to incur the cost of an MR test on that additional 5.5% than it is to do the full set of trial divisions.

The outcome of our analysis is a primality test whose performance on random, odd, 1024-bit inputs is on average 17% *faster* than the current OpenSSL test, but which guarantees that composites are identified with overwhelming probability $(1 - 2^{-128})$, no matter the input distribution. The downside is that, for inputs that are actually prime rather than random, our test is significantly slower than with OpenSSL's default settings (since we do 64 MR tests compared to the handful of tests used by OpenSSL). This is the price to be paid for a misuse-resistant API.

We then examine how our choice of primality test affects the performance of a crucial use case for primality testing, namely *generation* of random k-bit primes. OpenSSL 1.1.1c already includes code for this. It makes use of a sieving step to perform trial division at reduced cost across many candidates, obviating the need to perform per-candidate trial division internally to the primality test. OpenSSL avoids the internal trial division via the above-mentioned `do_trial_division` input to the primality test in OpenSSL. Since we do not allow such an input in our simplified primality testing API, a developer using our API would be (implicitly) forced to do trial division on a per candidate basis, potentially increasing the cost of prime generation. Moreover, our primality test may use many more rounds of MR testing than OpenSSL selects in this case, since our API does not permit the user to vary the

number of rounds according to the use case. However, for random prime generation, most candidates are rejected after just one MR test, and so the full cost of our test (trial division plus 64 rounds of MR testing) is only incurred once, when a prime is actually encountered. So we seek to understand the performance impact of plugging our new API and primality test into the existing OpenSSL prime generation code. We find that, for generation of random 1024-bit primes OpenSSL's prime generation code is 35-45% slower when using our primality test internally. For this cost, we gain an API for primality testing that is as simple as possible and where the test has strong security guarantees across *all* use cases.

We communicated our findings to the OpenSSL developers, and they have adopted our suggestions with only minor modifications: the forthcoming OpenSSL 3.0 (scheduled for release in Q4 of 2020) will include our simplified API for primality testing, and the OpenSSL codebase has been updated to use it almost everywhere (the exception is prime generation, which uses the old API in order to avoid redundant trial division). Moreover, OpenSSL will now always use our suggested primality test (64 rounds of MR) on all inputs up to 2048 bits, and 128 rounds of MR on larger inputs. This represents the first major reform of the primality testing code in OpenSSL for more than 20 years.

5.1.2 Related Work

The topic of API design for cryptography has a long history and connections to related fields such as usable security and API design for security more generally.

As early as 2002, Gutmann [70] identified the need to carefully define cryptographic APIs, recommending to "[p]rovide crypto functionality at the highest level possible in order to prevent users from injuring themselves and others through misuse of low-level crypto functions with properties they aren't aware of." This is precisely what we aim to do for primality testing in this chapter.

Later, Wurster and van Oorschot [164] (in the broader context of security) argued that attention should be focussed on those developers who produce core functionality used by other developers, e.g. producers of APIs. They identified the need to design

APIs which can be easily used in a secure fashion.

Green and Smith [68] extensively discuss the need for usable security APIs, and focus on cryptographic ones. They give an extensive list of requirements for good APIs, including: APIs should be easy to learn, even without cryptographic expertise; defaults should be safe and never ambiguous; APIs should be easy to use, even without documentation; APIs should be hard to misuse and incorrect use should lead to visible errors. These precepts have influenced our API design for primality testing.

Acar *et al.* [2] advocate for a research agenda for usable security and privacy research that focusses on developers rather than end users. This encompasses cryptography. Recent research related to this agenda and having a cryptographic focus includes [50, 52, 89, 1, 112, 111, 65].

Nonce-based Authenticated Encryption (AE), a primitive introduced by Rogaway [138], can be seen as an attempt to simplify the symmetric encryption API for developers, replacing the need to understand various requirements on IVs with the arguably simpler need to be able to supply unique (per key) inputs to an encryption algorithm. It has become the standard target for algorithm designers. However, as [27] showed, developers can accidentally misuse even this simplified API, with disastrous results for nonce-sensitive modes like AES-GCM. This motivated the development of misuse-resistant AE schemes, which attempt to preserve as much security as possible even when nonces are repeated. Prominent examples include SIV [139], Deoxys-II (part of the CAESAR competition final portfolio), and AES-GCM-SIV [69] (see also RFC 8452). Later authors identified the fact that developers may want an even higher-level API, for example a secure streaming channel like that provided by TLS [53, 127] or channels that tolerate some forms of reordering and repetition [30]; the mismatch between what developers want and what nonce-based AE can provide can lead to attacks, cf. [24].

Bernstein's design for DH key exchange on Curve25519 [19] deliberately presents a simple API for developers: public and private keys are represented by 32-byte strings, and the need for public key validation is avoided.

The NaCl crypto library [22] has provision of a simple API to developers as one of its primary aims. It gives the user a `crypto_box` function that encrypts and authenticates messages, with a simple API of the form: `c = crypto_box(m,n,pk,sk)`, where `m` is a message, `n` is a nonce, `pk` is the public key of the recipient and `sk` is the private key of the sender. Its security does rely on developers correctly handling nonces; we are unaware of reports of any misuse of this type. Some criticism of NaCl's approach, especially the way in which it breaks the developer's expected paradigm, can be found in [68].

There is an extensive literature on primality testing and generation, nicely summarised in [105, Chapter 4]. The state-of-the-art has not changed significantly since the publication of that book in 1996. On the other hand, as Chapter 3 showed, primality testing and generation as it is done in practice has many shortcomings. Our work can be seen as an effort to narrow the gap between the literature and its practical application.

5.1.3 Outline

The remainder of this chapter is organised as follows. In Section 5.2 we give further background on primality testing and detail the approach used in OpenSSL 1.1.1c. In Section 5.3 we describe four different candidate primality tests and analyse them theoretically and experimentally. Our chosen primality test (64 rounds of Miller-Rabin with a varied amount of trial division based upon the bit-size of the number being tested) emerges from this analysis as our preferred test. We then evaluate the performance of this chosen test in the use case of prime generation in Section 5.4. Section 5.5 briefly discusses how our test is being adopted in OpenSSL, while Section 5.6 contains our conclusions and avenues for future work.

5.2 Further Background

5.2.1 Primality Testing

In this chapter we will be discussing in depth the three primality tests introduced in Chapter 2. These are the Miller-Rabin, Lucas and Baillie-PSW tests. We shall also be discussing the details of the supplementary and preliminary tests (e.g. trial division).

5.2.1.1 Primality Testing in OpenSSL

Since we will extensively compare our primality test and its API with those of OpenSSL, we give a detailed description of the approach in OpenSSL 1.1.1c here.

OpenSSL's primality test is based mainly on the Miller-Rabin test as introduced in Section 2.4.2 and discussed in more detail in Section 3.3.1. The Miller-Rabin test is probabilistic, in that a t-round MR test using uniformly random bases declares any composite number to be composite with probability at least $1 - 4^{-t}$. Moreover, this bound is tight: there are composites which are not identified as being such over t rounds of testing with probability 4^{-t}. Such numbers, then, are worst-case adversarial inputs for the test. They are treated extensively in Chaper 3. On the other hand, the test *never* declares a prime to be composite.

The above discussion holds for any input n, no matter how it is chosen. When n is a uniformly random odd k-bit integer, much better performance can be assured. For example, a result of [41] assures that the probability $p_{k,1}$ that a composite n chosen in this way passes one round of random-base MR testing is bounded by $k^2 4^{2-\sqrt{k}}$. Thus, for $k = 1024$, we have $p_{k,1} \leq 2^{-40}$. Using more precise bounds from [41], this can be improved to $p_{k,1} \leq 2^{-42.35}$. These bounds are what motivates the rather small numbers of rounds of MR testing in the default setting in OpenSSL's primality test.

OpenSSL provides two functions for primality testing: `BN_is_prime_ex` and `BN_is_prime_fasttest_ex`, both in file `bn_prime.c`. The core part of the code is in the

k	t	λ (bits)
$k \geq 3747$	3	192
$k \geq 1345$	4	128
$k \geq 476$	5	80
$k \geq 400$	6	80
$k \geq 347$	7	80
$k \geq 308$	8	80
$k \geq 55$	27	64
$k \geq 6$	34	64

Table 5.1: The default number of rounds t of Miller-Rabin performed by OpenSSL 1.1.1c when testing k-bit integers determined by the function `BN_prime_checks_for_size` and the associated bits of security λ.

second of these, while the first simply acts as a wrapper to this function that forces omission of trial division. The second function call has the form:

```
int BN_is_prime_fasttest_ex(const BIGNUM *w, int checks,
BN_CTX *ctx_passed, int do_trial_division, BN_GENCB *cb)
```

Here, `w` is the number being tested. The option to do trial division is defined via the `do_trial_division` flag. When set, the function will perform trial division using the first 2047 odd primes (excluding 2), with no gcd optimisations (the code also separately tests whether the number being tested is equal to 2 or 3, and whether it is odd). After this, the function calls `bn_miller_rabin_is_prime` to invoke the MR testing with pseudo-random bases. The number of MR rounds is set using the argument `checks`. When `checks` is set to `BN_prime_checks`, a value that defaults to zero, then the number of MR rounds is chosen such that the probability of the test declaring a random composite number n with k bits as being prime is at most $2^{-\lambda}$, where λ is the security level that a $2k$-bit RSA modulus should provide. Thus, the number of MR rounds performed is based on the bit-size k, as per Table 5.1. The entries here are based on average case error estimates taken from [105], which in turn references [41].[4]

[4]One might note how Table 5.1 differs from that shown in Table 3.4 of Section 3.3.1. This is due to the natural development of OpenSSL between the version 1.1.1-pre6 in August 2018 and OpenSSL 1.1.1c of May 2019 - for more information on this, see Section 1.2.2.

5.2.2 Prime Generation

A critical use case for primality testing is prime generation (e.g. for use in RSA keys). The exact details of the algorithms used vary across implementations, but the majority follow a simple technique based on first generating a random initial candidate n of the desired bit size k, possibly setting some of its bits, then doing trial division against a list of small primes, before performing multiple rounds of primality testing using a standard probabilistic primality test such as the MR test. If the trial division reveals a factor or the MR test fails, then another candidate is generated. This can be a fresh random value, but more commonly, implementations add 2 to the previous candidate n. This allows an important optimisation: if a table of remainders for the trial divisions of n is created in the first step, then this table of remainders can be quickly updated for the new candidate $n + 2$. Fresh divisions can then be avoided – one just needs to inspect the updated table of remainders. We refer to this procedure as *trial division by sieving* or just *sieving*. It is, of course, much more efficient than performing trial divisions anew for each candidate. Note that this approach leads to a slightly non-uniform distribution on primes: primes that are preceded by a long run of composites are more likely to result from it than primes that are close to their preceding primes. However, it is known that the deviation from the uniform distribution is small [31].

5.2.2.1 Prime Generation in OpenSSL

OpenSSL adopts the above high-level procedure, with one important difference. The code is found in `BN_generate_prime_ex` in file `bn_prime.c`. The function call has the following form:

```
int BN_generate_prime_ex(BIGNUM *ret, int bits, int safe, const BIGNUM
*add, const BIGNUM *rem, BN_GENCB *cb)
```

Here `bits` is the desired bit-size, `safe` is a flag that, when set, asks the function to produce a safe prime $p = 2q + 1$, and `add` and `rem` allow the callee to set additional conditions on the returned prime. We will ignore `safe`, `add` and `rem` in our further

work; an analysis of how they affect prime generation when using our primality test is left to future work.

The initial steps are performed together in a separate function called `probable_prime`. A cryptographically strong pseudo-random number is first generated by `BN_priv_rand`. The two most significant bits and the least significant bit are then set to ensure the resulting candidate n is odd and of the desired bit-size. This number is then sieved using a hard-coded list of the first 2047 odd primes $p_2, \ldots, p_{2048}$, so $p_1 = 2, p_2 = 3, \ldots, p_{2048} = 17863$. If a candidate passes the sieving stage, it is tested for primality by `BN_is_prime_fasttest_ex`. This function carries out the default number of Miller-Rabin rounds, as per Table 5.1. Trial division is omitted by setting the `do_trial_division` flag in the function call. This is because trial division has already been carried out externally via sieving. This exploits the complexity of the OpenSSL API for primality testing to gain performance, an option not available if a simplified API is desired (as we do). Importantly, if the MR tests fail, then instead of going to the next candidate that passes sieving, a fresh, random starting point is selected and the procedure begins again from the start.

5.3 Construction and Analysis of a Primality Test With a Misuse-resistant API

We now propose how to construct a performant primality test with a misuse-resistant API. Our design goal is to ensure good performance in the most important use cases (malicious input testing, prime generation) while still maintaining strong security. At the same time, we want the simplest possible API for developers: a single input n (the number being tested) and single a 1-bit output (0 for composite, 1 for probably prime).

We propose four different primality testing functions, all built from the algorithms described in Chapter 2. The first of these follows OpenSSL with its default settings, and we name this Miller-Rabin Average Case (MRAC). It provides a baseline for analysis and comparison. The second and third use 64 and 128 rounds of MR testing, respectively. We name them MR64 and MR128. The fourth uses the Baillie-PSW test, and we name it BPSW for short. For each of these four options, we provide

an assessment (both by analysis and by simulation) of its security and performance when considering random composite, random prime, and adversarially generated composite inputs. We also consider the influence of trial division on each test's performance. For concreteness, throughout we focus on the case of 1024-bit inputs, but of course the results are easily extended to other bit-sizes.

5.3.1 Miller-Rabin Average Case (MRAC)

The first test we introduce, MRAC, is a reference implementation of OpenSSL's primality test, as per the function `BN_is_prime_fasttest_ex` described in Section 5.2.1.1 with input `checks` set to `BN_prime_checks`, so that the number of MR rounds performed is based on the bit-size k, as per Table 5.1. Recall that this function either does no trial division or does trial division with the first 2047 odd primes. Of course, this test is quite unsuitable for use in general, because it performs badly on adversarial inputs: Chapter 3 showed that it has a worst case false positive rate of $1/2^{2t}$ where for example $t = 5$ for 1024-bit inputs. On the other hand, it is designed to perform well on random inputs.

5.3.1.1 MRAC on Random Input

We now consider the expected number of MR rounds performed when receiving a random 1024 bit odd input. For now, we ignore the effect of trial division. The probability that a randomly chosen odd k-bit integer is prime is $q_k := 2/\ln(2^k)$ by standard estimates for the density of primes (for $k = 1024, q_k \approx 1/355$). In this case MRAC will do t MR rounds, as per Table 5.1. Otherwise, for composite input, up to t rounds of MR testing will be done. One could use the bounds from [41] to obtain bounds on the expected number of MR rounds that would be carried out on composite input. However, for numbers of cryptographic size (e.g. $k = 1024$ bits), to a very good approximation, the number needed is just 1, since with very high probability, a single MR test is sufficient to identify a composite (recall that the probability that a single round of MR testing fails to identify a 1024-bit composite is less than 2^{-40}). From this, one can compute the expected number of rounds needed for a random, odd input: it is approximately the weighted sum $t \cdot q_k + 1 \cdot (1 - q_k) =$

$1 + (t-1)q_k$. For $k = 1024$, we have $t = 5$ and $q_k = 0.0028$, and this expression evaluates to 1.026.

5.3.1.2 MRAC on Random Input with Trial Division

Now we bring trial division into the picture. Its overall effectiveness will be determined by the collection of small primes in the list $P = \{p_1, p_2, \ldots, p_r\}$ used in the process (where we assume all the p_i are odd) and the relative costs of MR testing and trial division (about 800:1 in our experiments).

For random odd inputs, the fraction $\sigma(P)$ of non-prime candidates that are removed by the trial division by the primes in P can be computed using the formula:

$$\sigma(P) = 1 - \prod_{i=1}^{r} \left(1 - \frac{1}{p_i}\right). \tag{5.1}$$

This follows easily by noting that a fraction $1 - \frac{1}{p_i}$ of integers are not divisible by p_i, so the probability that a randomly sampled integer is *not* divisible by any of the p_i is $\prod_{i=1}^{r} \left(1 - \frac{1}{p_i}\right)$, and hence the probability that a randomly sampled odd integer is divisible by at least one p_i is $\sigma(P)$. In turn, this means that any candidate that passes the trial division stage is $1/(1-\sigma(P))$ times more likely to be a prime than an odd candidate of equivalent bit-size chosen at random (this is because a fraction $1 - \sigma(P)$ of integers remain after sieving, and all primes survive sieving).

But simply adding more primes to the list P is not necessarily effective: fewer additional composites are removed at a fixed cost (one additional trial division per prime), and eventually it is better to move on to a more heavyweight test (such as rounds of MR testing). Moreover, from inspecting the formula for $\sigma(P)$, it is evident that, for a given size r of set P (and hence a given cost for trial division), it is better to set P as containing the r smallest odd primes (including 2 is not useful as the input n is already assumed to be odd). Henceforth, we assume that when P is of size r, then it consists of the first r odd primes. We write σ_r in place of $\sigma(P)$ in this case. Using Mertens' theorem, we can approximate σ_r as follows:

$$\sigma_r \approx 1 - 2e^{-\gamma}/\ln(p_r).$$

where $\gamma = 0.5772\ldots$ is the Euler-Mascheroni constant.

As an example, `BN_is_prime_fasttest_ex` in OpenSSL performs trial division on the first 2047 odd primes (ending at $p_{2047} = 17863$). As shown in Figure 5.1, using the first $r = 2047$ primes gives a value of $\sigma_{2047} = 0.885$. This is only a little larger than using, say, the $r = 128$ smallest odd primes yielding $\sigma_{128} = 0.831$.

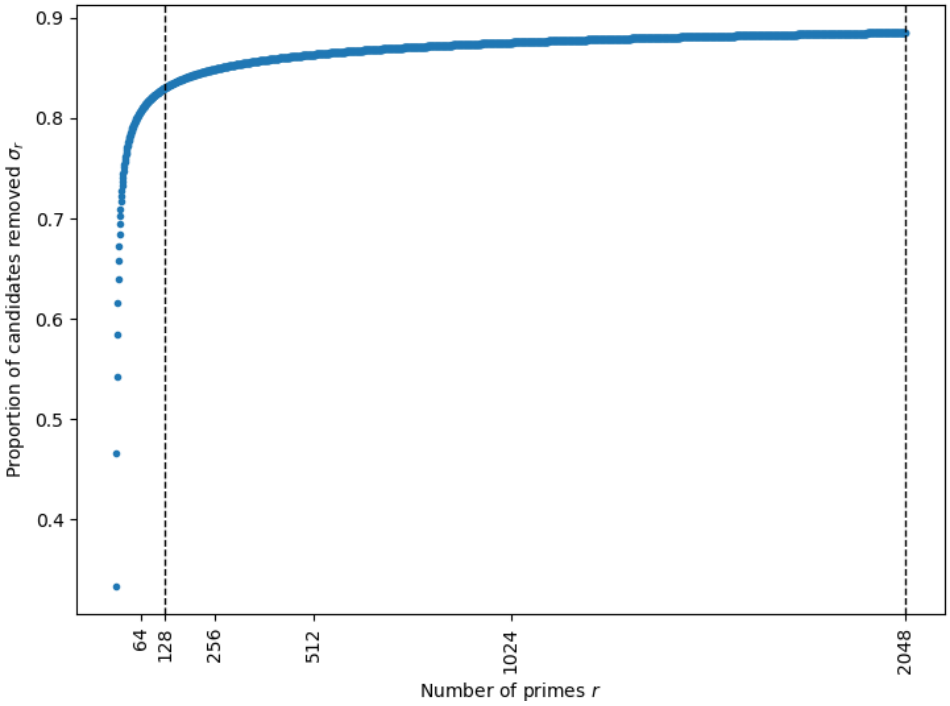

Figure 5.1: Proportion of candidates removed by trial division, σ_r, as a function of r, the number of primes used.

Now we build a cost model for MRAC including trial division. This will also be applicable (with small modifications) for our other tests.

Let C_i denote the cost of a trial division for prime p_i and let C_{MR} denote the cost of a single MR test.[5] Then the total cost of MRAC on random *prime* k-bit inputs is:

$$\sum_{i=1}^{r} C_i + t \cdot C_{MR} \tag{5.2}$$

since the test then always performs all r trial divisions (assuming k is large enough) and all t MR tests. For random, odd *composite* inputs, the average cost is approxi-

[5] In practice, we could set C_i to be a constant C_{TD} for the range of i we are interested in, but using a more refined approach is not mathematically much more complex.

mately:

$$\sigma_1 \cdot C_1 + (\sigma_2 - \sigma_1) \cdot (C_1 + C_2) + \ldots + (\sigma_r - \sigma_{r-1}) \cdot (C_1 + \cdots + C_r) \\ +(1 - \sigma_r) \cdot (\sum_{i=1}^{r} C_i + C_{MR}).$$

This is because a fraction σ_1 of the composites are identified by the first trial division, a further fraction $\sigma_2 - \sigma_1$ are identified after 2 trial divisions, etc, while a fraction $(1 - \sigma_r)$ require all r trial divisions plus (roughly) 1 round of MR. Here we assume that the MR test performs in the same way on numbers after trial division as it does before. After some manipulation, this last expression can be simplified to:

$$\sum_{i=1}^{r}(1 - \sigma_{i-1}) \cdot C_i + (1 - \sigma_r) \cdot C_{MR} \tag{5.3}$$

where we set $\sigma_0 = 0$. This expression can be simplified further if we assume that the C_i are all equal to some C_{TD} (a good approximation in practice), and apply Mertens' theorem again. For details, see the equivalent analysis in [102].

From expressions (5.2) and (5.3), the expected cost for random, odd, k-bit input can be easily computed via a weighted sum $(q_k)(5.2) + (1 - q_k)(5.3)$. However, the cost is dominated by expression (5.3) for the composite case. From (5.3), the futility of trial division with many primes is revealed: adding a prime by going from r to $r+1$ on average adds a term $(1 - \sigma_r) \cdot C_{r+1}$, but only decreases by a fraction $\sigma_{r+1} - \sigma_r$ the term in front of C_{MR}. As can be seen from Figure 5.1, when r is large, $1 - \sigma_r$ is around 0.1, while $\sigma_{r+1} - \sigma_r$ becomes very small. So each increment in r only serves to increase the average cost by a fraction of a trial division (and with the cost of trial division increasing with r).

Figure 5.2 shows a sample (theoretical) plot of the average cost of MRAC as a function of r for $k = 1024$. This uses as costs $C_{TD} = 0.000371$ms and $C_{MR} = 0.298$ms obtained from our experiments (reported below) for $k = 1024$ and the weighted sum of expressions (5.2), (5.3). This curve broadly confirms the analysis of [102] which suggests setting $p_r = C_{MR}/C_{TD}$ to minimise the running time of primality testing with trial division; here we obtain $C_{MR}/C_{TD} \approx 800$, corresponding to $r \approx 140$.[6]

[6]The analysis of [102] technically applies to prime generation, but ignores certain terms in such a way as to actually analyse the cost of primality testing of composite numbers. In this sense, it is only valid when the cost of primality testing for *prime* inputs can be ignored compared to the case of composite inputs; this is not the case in general, but is a reasonable approximation for MRAC.

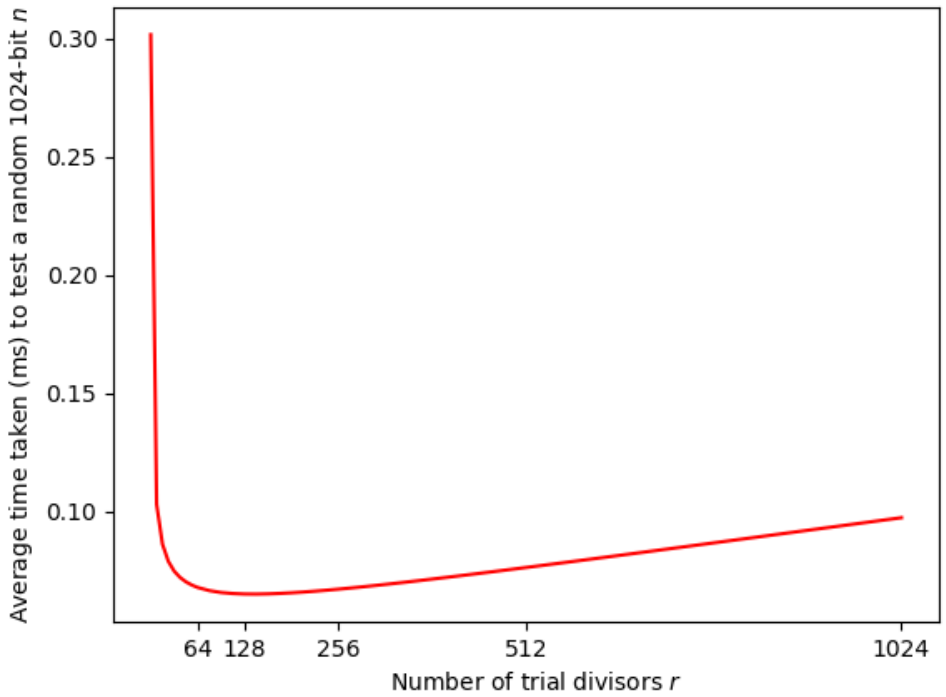

Figure 5.2: A plot of the theoretical running time of MRAC as a function of r, the number of primes r used in trial division for $k = 1024$, using $C_{TD} = 0.000371$ms and $C_{MR} = 0.298$ms obtained from our experiments.

5.3.1.3 MRAC on Adversarial Input

Recall from Chapter 3 that worst-case adversarial inputs can fool random-base MR testing with probability 1/4 per round. The expected number of rounds needed to identify such inputs as composite is then 1.33. However, with t rounds of testing, MRAC will fail to identify such composites as being so with probability $1/2^{2t}$ (and will indicate that the input was prime). Note that this analysis is unaffected by trial division, since the adversarial inputs used have no small primes factors – the trial division just increases the running time of the test.

5.3.2 Miller-Rabin 64 (MR64)

Next we consider trial division followed by up to 64 rounds of MR testing with random bases (the test will exit early if a base that is a witness to compositeness of the input n is found). We refer to this test as MR64. By design, this test guarantees

a failure probability of at most 2^{-128}, no matter the input distribution, so it offers robust security guarantees without the user needing to understand the context of the test (i.e. whether the test is being done with adversarial inputs or not).

5.3.2.1 MR64 on Random Input

As for MRAC, for a random, odd composite, k-bit input, the expected number of rounds of MR testing (without trial division) is very close to 1. On the other hand, for prime, k-bit input, the number of rounds is exactly 64. This enables the average cost without trial division on random, odd, k-bit input to be computed: it is approximately given by the weighted sum

$$(64 \cdot q_k + 1 \cdot (1 - q_k)) \cdot C_{MR} = (1 + 63q_k) \cdot C_{MR}$$

For $k = 1024$, we again have $q_k = 2/\ln(2^k) = 0.0028$, and this sum evaluates to $1.18C_{MR}$, about 17% higher than MRAC for the same input distribution.

5.3.2.2 MR64 on Random Input with Trial Division

Following the analysis for MRAC, we can compute the cost of MR64 on random, *prime*, k-bit input as:

$$\sum_{i=1}^{r} C_i + 64 \cdot C_{MR} \tag{5.4}$$

since here all trial divisions are performed, together with 64 rounds of MR testing. For random, odd, *composite* input with r-prime trial division, the expected cost is very close to that of MRAC with the same r, since whenever MR testing is invoked, almost always one round suffices. As for the case of MR64 without trial division, it is the prime inputs that make the cost difference here: they involve 64 rounds of MR testing instead of the (close to) 1 needed for composite inputs. Again, a theoretical prediction for random, odd input can be made by combining the expressions for odd, composite and prime input using a weighted sum. This sum is $(q_k)(5.4)+(1-q_k)(5.3)$, differing only from the cost model for MRAC by the term representing the cost of testing prime input. Figure 5.3 shows the theoretical curve for MR64 as compared to MRAC (using costs $C_{TD} = 0.000371$ms and $C_{MR} = 0.298$ms for $k = 1024$ as before).

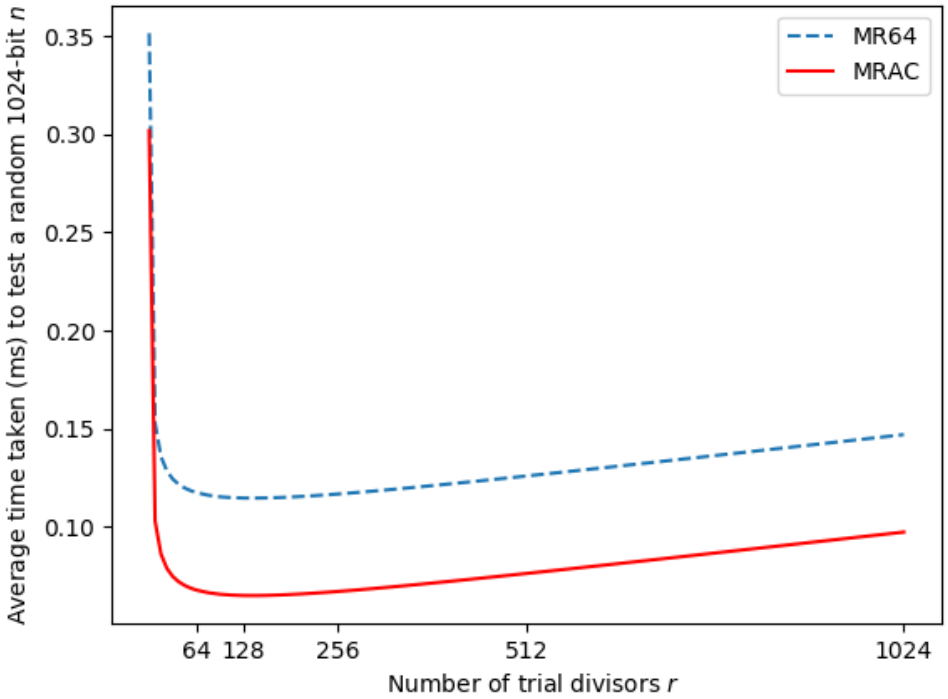

Figure 5.3: Comparing the theoretical running time of MR64 and MRAC as a function of r (the number of primes r used in trial division) for $k = 1024$, using $C_{TD} = 0.000371$ms and $C_{MR} = 0.298$ms obtained from our experiments.

5.3.2.3 MR64 on Adversarial Input

By design, the MR64 test will fail to identify a worst-case adversarial input as a composite with probability at most 2^{-128}, this after 64 rounds of MR testing. The *expected* number of rounds needed to successfully classify such inputs is again 1.33.

5.3.3 Miller-Rabin 128 (MR128)

This test is identical to MR64, but up to 128 rounds of MR testing are invoked. The intention is to reduce the false positive rate from 2^{-128} to 2^{-256}. The analysis is almost identical to that for MR64, replacing 64 by 128 where it appears in the relevant formulae. We include it for comparison purposes and because the OpenSSL documentation does target 256 bits of security when testing very large numbers

(larger than 6394 bits in size[7]). The headline figure for this test is its expected cost (without trial division) of $(1 + 127q_k) \cdot C_{MR}$, equating to $1.36 \cdot C_{MR}$ on random, odd, 1024-bit inputs, roughly 35% higher than MRAC at the same input size.

5.3.4 Baillie-PSW (BPSW)

The final test we consider is the Baillie-PSW test. Recall that this is the combination of a single Miller-Rabin test to base 2, with a Lucas test using Selfridge's Method A to select D. If the input n we are testing is a perfect square, then there does not exist a valid choice of D (see Section 2.4.4). So we must decide upon a point to test for this. Baillie and Wagstaff [15] show that, when n is not square, the average number of D values that need to be tried until a suitable one is found is 1.78. We choose to run a test to check if n is a perfect square only after 7 unsuccessful attempts to select D. This choice is inspired by other implementations [96] and the fact that, if n is a random non-square, then the probability of failing 7 times is extremely unlikely. This provides a balance between the relatively cheap process of testing a choice of D with the more expensive test for n being a perfect square. We perform the Miller-Rabin part of the test first, since it is the more efficient of the two techniques, omitting the Lucas test early if this indicates compositeness. We then search for D using Selfridge's Method A, using it to carry out a Lucas test if found. We abort the search for D after 7 attempts and then test n for being a perfect square. If this test fails, we revert to searching for a suitable D and then perform the Lucas test when one is eventually found.

Algorithm 3 gives example pseudocode for the process of selecting the parameter D and performing the Baillie-PSW test in the manner described above. The detail of the pseudocode for the Miller-Rabin test, Lucas test, Jacobi symbol calculation, and perfect square test are omitted, as these are discussed within the preliminary material in Chapter 2. However, a full reference implementation (including all of these functions) of the Baillie-PSW test used for this work is included in Appendix B.1.

[7] See the man page `https://www.openssl.org/docs/man1.1.0/man3/BN_is_prime_fasttest_ex.html` and code documentation `https://github.com/openssl/openssl/blob/fa4d419c25c07b49789df96b32c4a1a85a984fa1/include/openssl/bn.h#L159`.

Algorithm 3 The Baillie-PSW test on n

Input a positive integer n.

Output the result of the B-PSW test on n. Returns 1 if the test indicates that n is prime and 0 if n is composite.

Step 1: perform the Miller-Rabin test on n. Here $\mathsf{MillerRabin}(a, n)$ is a function that performs a single round of Miller-Rabin on n to base a, returning 1 if n is a probable prime and 0 if n is composite.

1: $a \leftarrow 2$
2: **if** $\mathsf{MillerRabin}(a, n) == 0$ **then**
3: $\quad res \leftarrow 0$
4: $\quad$ go to Step 4
5: **end if**

Step 2: find the parameter D for the Lucas test. Here $\mathsf{JacobiSymbol}(D, n)$ is a function that returns the Jacobi symbol $\left(\frac{D}{n}\right)$ and $\mathsf{PerfectSquare}(n)$ is a function that returns 1 if n is a perfect square and 0 otherwise.

6: $D \leftarrow 5$
7: **while** $\mathsf{JacobiSymbol}(D, n) \neq -1$ **do**
8: $\quad$ **if** $D < 0$ **then**
9: $\quad\quad D \leftarrow abs(D) + 2$ $\quad$ # where $abs(x)$ is the absolute value of x.
10: $\quad$ **else**
11: $\quad\quad D \leftarrow -(D + 2)$
12: $\quad$ **end if**
13: $\quad$ **if** $D == -19$ **then** # we have 7 failed attempts at finding D with $\left(\frac{D}{n}\right) = -1$.
14: $\quad\quad$ **if** $\mathsf{PerfectSquare}(n) == 1$ **then**
15: $\quad\quad\quad res \leftarrow 0$
16: $\quad\quad\quad$ go to Step 4
17: $\quad\quad$ **end if**
18: $\quad$ **end if**
19: **end while**

Step 3: perform the Lucas test on n. Here $\mathsf{Lucas}(P, Q, D, n)$ performs a Lucas test on n, returning 1 if the test indicates that n is prime and 0 if n is composite.

20: $P \leftarrow 1$ $\quad$ # the remaining parameters for Selfridge's method are set.
21: $Q \leftarrow (1 - D)/4$
22: **if** $\mathsf{Lucas}(P, Q, D, n) == 0$ **then**
23: $\quad res \leftarrow 0$
24: $\quad$ go to Step 4
25: **end if**
26: $res \leftarrow 1$ $\quad$ # if we have reached this point, n has passed the B-PSW test.

Step 4: output result.

27: Return res

5.3.4.1 BPSW on Random Input

The analysis without trial division is much like that of MRAC, assuming that MR with a fixed base 2 performs as well as MR with a random base when the number being tested is uniformly random. For prime inputs, the average cost is $C_{MR} + C_L$, where C_L is average the cost of doing the Lucas part of the test (and any tests of squareness); for composite inputs, the cost is roughly C_{MR} since the MR test catches the vast majority of composites. The performance on random inputs is the weighted sum of these, as usual. In our implementation, the average for C_L for 1024-bit inputs is equal to $17.04 \cdot C_{MR}$ (5.078ms compared to 0.298ms on average for 1024-bit inputs, based on 2^{20} trials). Overall, then, this test has an expected cost (without trial division) of $1.05 \cdot C_{MR}$ on random, odd, 1024-bit inputs, roughly 4% more than MRAC.

The analysis with trial division is again similar to that for MRAC: when the input is prime, the average cost is $\sum_{i=1}^{r} C_i + C_{MR} + C_L$, while when the input is composite, it is of the same form as in (5.3) (where we are able to omit a term C_L under the assumption that the base 2 MR test is effective in detecting composites). We omit further detail.

5.3.4.2 BPSW on Adversarial Input

It is relatively easy to construct composites passing a base 2 MR test. For example, integers of the form $(2x + 1)(4x + 1)$ with each factor a prime have a roughly 1 in 4 chance of doing so (see Section 3.2.1.1 for further discussion). Such inputs are highly likely to be detected by the Lucas part of the BPSW test, so the cost of BPSW on such inputs would be $\sum_{i=1}^{r} C_i + C_{MR} + C_L$. However, we do not know if such numbers are worst-case adversarial inputs for BPSW, and indeed, we cannot rule out the existence of BPSW pseudoprimes, that is, composites which are declared probably prime by the test. Recall that Pomerance [133] has given heuristic evidence that there are infinitely many such pseudoprimes. Perhaps the smallest is beyond the bit-size we care about in cryptographic applications, but we cannot be sure. Note also that such a pseudoprime, if it can be found, would always fool the BPSW test (because the choice of parameters used within the test is deterministic). This

is in sharp contrast to MR64 and MR128, where we can give precise bounds on the false positive rate of the tests. We consider this, along with the relative complexity of implementing the BPSW test, to be a major drawback.

5.3.5 Experimental Results

Having described our four chosen primality tests and given a theoretical evaluation of them, we now turn to experimental analysis. This analysis gives us a direct comparison with the current approach of OpenSSL (MRAC with trial division either off or based on 2047 primes). It also allows us to study how the Baillie-PSW test performs against Miller-Rabin testing in practice, something that does not appear to have been explored before. We focus initially on testing 1024-bit numbers to avoid deluging the reader with data; results for other bit-sizes are presented later in the section.

5.3.5.1 Random Input

Our results for random, odd, 1024-bit inputs to the tests are shown in Tables 5.2 and 5.3. We worked with 2^{25} inputs, produced using OpenSSL's internal random number generator. All timings are in milliseconds, and are broken down into results for composite inputs, inputs that were declared prime, and overall results. We also report results for different amounts of trial division — none, $r = 128$ (which, from our theoretical analysis above, we consider to be a sensible amount of trial division for 1024-bit inputs) and $r = 2047$ (as in OpenSSL). All results were obtained using a single core of an Intel(R) Xeon(R) CPU E5-2690 v4 @ 3.20GHz processor, with code written in C using OpenSSL 1.1.1c (May 2019) for big-number arithmetic and basic Miller-Rabin functionality.

Of the 2^{25} random, odd, 1024-bit numbers that we generated, 94947 were prime. This is closely in line with the estimated $q_{1024} \times 2^{25} \approx 94548$ given by the usual density estimate.

The results in Table 5.2 are broadly in-line with our earlier theoretical analysis.

Some highlights:

- MRAC is fast overall, but with $r = 2047$, OpenSSL is doing far too much trial division on 1024-bit inputs. Much better performance could be achieved for this input size in OpenSSL by setting $r = 128$ (more than 2x speed-up overall can be gained).
- MR64 is 8-9 times slower than MRAC on prime input, reflecting the many more rounds of MR testing being done in MR64.
- MR128 is roughly twice as slow as MR64 on prime input (reflecting the doubling of rounds of MR testing). On random input, the gap between MR64 and MR128 is not so large (because most composites are identified by trial division or after just one round of MR testing).
- BPSW is quite competitive with MRAC overall and only 2-3 times slower for prime input. This is because the Lucas test part of BPSW is expensive but rarely invoked for random input, but always done for prime input.
- Based on overall figures, MR64 with $r = 128$ outperforms MRAC with $r = 2047$ (as used in OpenSSL) by 17% on 1024-bit input. This indicates that, by tuning parameters carefully, it is possible to obtain improved performance over the current approach used in OpenSSL whilst enjoying strong security across all use cases (i.e. a guaranteed false positive rate of 2^{-128}). Even MR128 with $r = 128$ is not far behind MRAC with $r = 2047$ on overall figures at this input size.

Further improvements in running time can be obtained by fine-tuning the value of r on a per test basis, and according to input size. Importantly, the latter is feasible even with a simple API (and indeed seems to be the only general, input-dependent optimisation possible). To illustrate this, we show in Figure 5.4 the average running times for MRAC and MR64 on random, odd, 1024-bit input for varying r. The figure also shows the theoretical curves obtained previously. There is an excellent agreement between the experimental data and the curves obtained from the model. In both cases, the curve is quite flat around its minimum, but we see that using $r = 128$ gives close to optimal performance for this value of $k = 1024$. The figure also illustrates that using large amounts of trial division (as per OpenSSL) harms

r	Declared Composite			
	MRAC	MR64	MR128	BPSW
0	0.312	0.313	0.312	0.302
128	0.063	0.063	0.063	0.061
2047	0.135	0.134	0.134	0.133
r	Declared Prime			
	MRAC	MR64	MR128	BPSW
0	1.50	19.1	38.1	5.39
128	1.55	19.1	38.2	5.44
2047	2.26	19.8	38.9	6.15
r	Overall			
	MRAC	MR64	MR128	BPSW
0	0.315	0.366	0.419	0.316
128	0.067	0.117	0.170	0.077
2047	0.141	0.190	0.244	0.150

Table 5.2: The mean running time (in ms) for each test when testing MRAC, MR64, MR128 and BPSW for random 1024-bit, odd inputs and various amounts of trial division (r). Broken down by input primality. Results based on 2^{25} trials.

r	Declared Composite			
	MRAC	MR64	MR128	BPSW
0	0.007	0.007	0.007	0.006
128	0.133	0.133	0.133	0.129
2047	0.338	0.338	0.337	0.335
r	Declared Prime			
	MRAC	MR64	MR128	BPSW
0	0.028	0.359	0.716	0.099
128	0.028	0.358	0.717	0.097
2047	0.032	0.363	0.720	0.104
r	Overall			
	MRAC	MR64	MR128	BPSW
0	0.064	1.00	2.01	0.270
128	0.154	1.02	2.02	0.312
2047	0.356	1.10	2.08	0.462

Table 5.3: The standard deviation of the running time (in ms) for each test when testing MRAC, MR64, MR128 and BPSW for random 1024-bit, odd inputs and various amounts of trial division (r). Results based on 2^{25} trials.

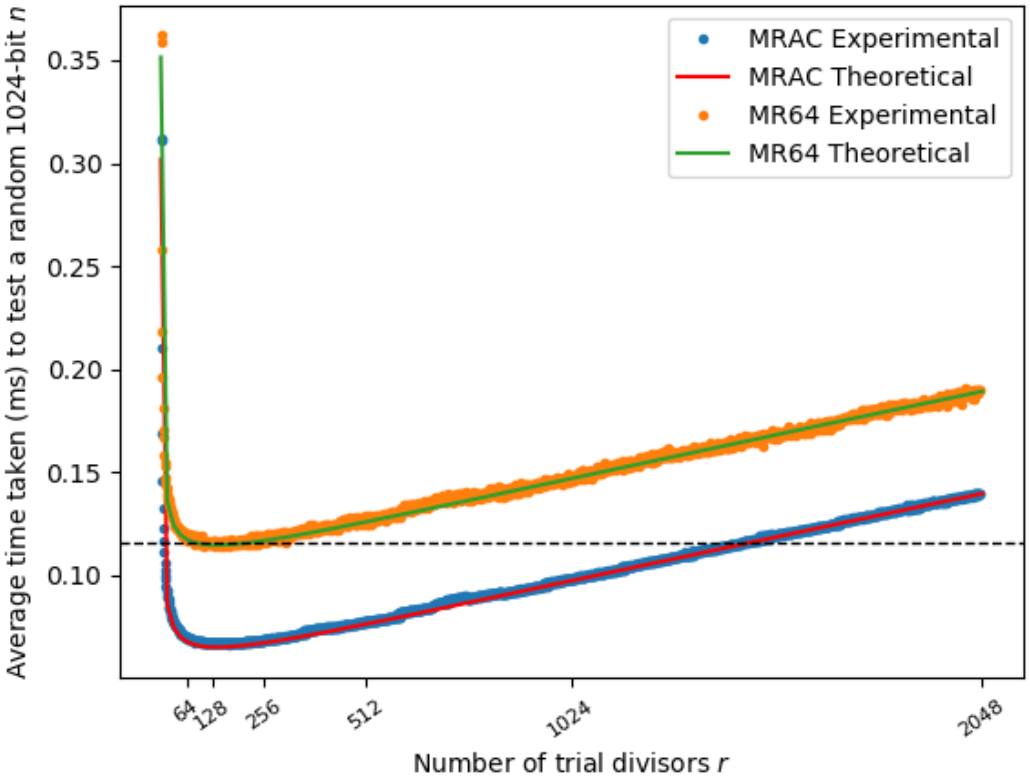

Figure 5.4: Experimental and theoretical performance of MRAC and MR64 on random, odd, 1024-bit input for varying amounts of trial division, r. The horizontal dashed line represents the minimum of the average running time of MR64 across all choices of r. This gives a visual representation of the comparison between MR64 with $r = 128$ and MRAC with $r = 2047$.

performance for this input size, as was also explained theoretically in Section 5.3.1. Specifically, OpenSSL uses $r = 2047$, putting its performance with default settings (MRAC) well above the minimum obtainable with MR64 with a carefully tuned choice of r.

5.3.5.2 Adversarial Input

To bring into sharp relief the failings of MRAC as a general-purpose primality test, we generated a set of 2^{20} 1024-bit composites of the form $n = (2x + 1)(4x + 1)$ in which the factors $2x + 1$, $4x + 1$ are both prime. Numbers of this special form

Rounds	MRAC	MR64
1	787054	786765
2	196110	196268
3	49167	49305
4	12157	12103
5	4088	3129
6	–	776
7	–	169
8	–	44
9	–	13
10	–	4

Table 5.4: Number of rounds of MR testing needed to identify as composite 1024-bit numbers of the form $n = (2x+1)(4x+1)$ with $2x+1$, $4x+1$ prime from an initial set of 2^{20} candidates. MRAC only performs 5 rounds of MR testing for this bit-size and failed to identify exactly 1000 candidates.

are known to pass random-base MR tests with probability 1/4. We then put these n through our MRAC and MR64 tests without trial division,[8] tracking how many rounds of MR were used on each input by each test. Table 5.4 shows the results. MR64 needed a maximum of 10 rounds of MR testing to correctly classify all the inputs, while MRAC, using only 5 rounds of MR for inputs of this size, incorrectly classified exactly 1000 of the inputs. This performance is in-line with expectations, as the expected number of misclassifications is $2^{20} \times (1/4)^5 = 2^{10}$.

5.3.6 Other Bit Sizes

So far in our experimental evaluation, we have focussed on $k = 1024$, i.e. testing of 1024-bit inputs. We have carried out similar testing also for $k = 512, 2048, 3072$. Figures 5.5, 5.6 and 5.7 show these additional results for the MRAC and MR64 tests, focussing on the effect of varying r on running time. Notice the characteristic "hockey-stick" shape of the curves in all the figures.

In each figure, the dashed horizontal time highlights the minimum running time for MR64. Notably, for $k = 512$, this is significantly lower than MRAC with $r = 2047$ (as in OpenSSL). We saw the same effect for $k = 1024$ in Figure 5.4. For $k = 2048$,

[8] Including trial division would not change the results.

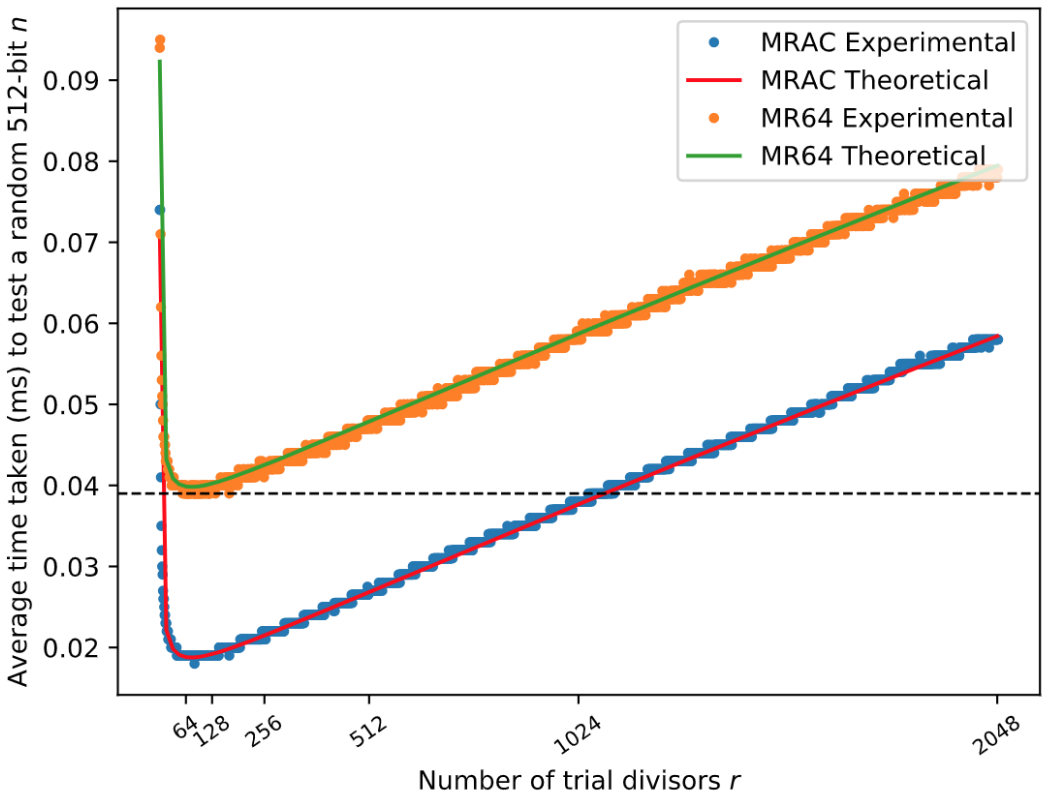

Figure 5.5: Experimental and theoretical performance of MRAC and MR64 on random, odd, 512-bit input for varying amounts of trial division, r.

MR64 with the best choice of r is slightly slower than MRAC with $r = 2047$ (but still competitive). For $k = 3072$, the influence of r on running time is quite small, and MRAC consistently comes out ahead of MR64 (but recall that MRAC is unsafe for maliciously chosen inputs).

These experiments confirm our earlier observation: the choice of r, the amount of trial division, can have a significant effect on running time of primality tests, and should be taken into account when selecting a test.

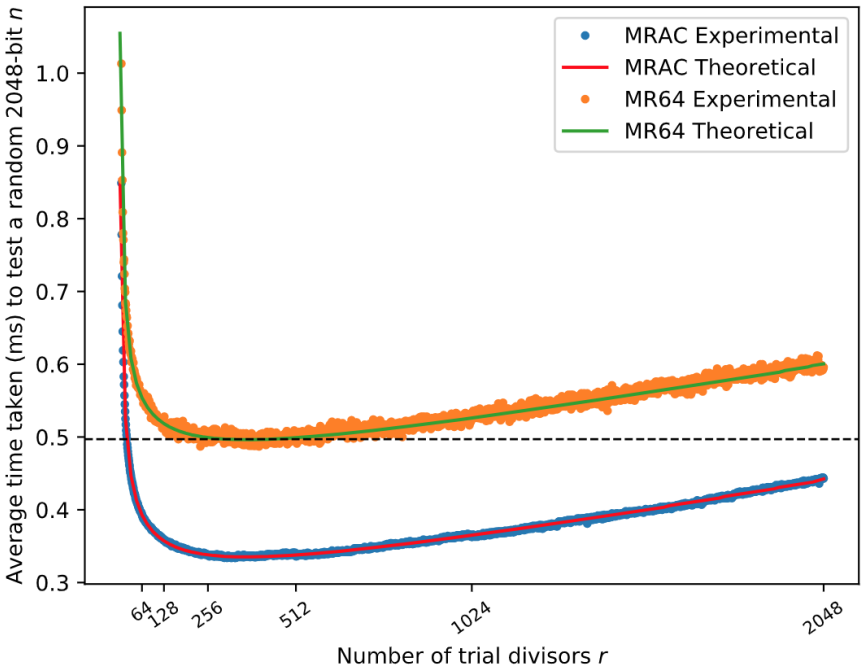

Figure 5.6: Experimental and theoretical performance of MRAC and MR64 on random, odd, 2048-bit input for varying amounts of trial division, r.

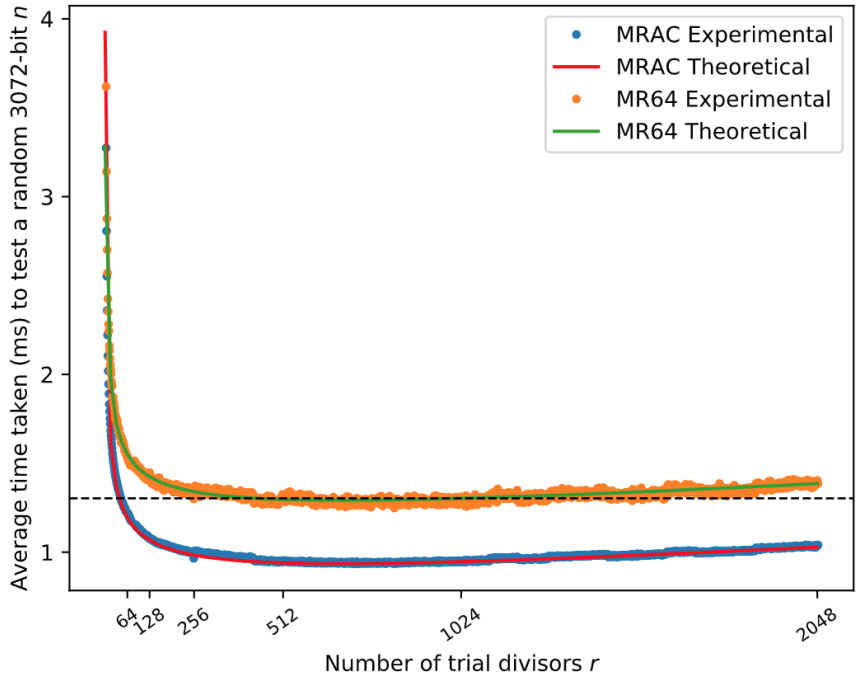

Figure 5.7: Experimental and theoretical performance of MRAC and MR64 on random, odd, 3072-bit input for varying amounts of trial division, r.

5.3.7 Selecting a Primality Test

We select MR64 with the amount of trial division, r, depending on the input size as our preferred primality test. Our reasons are as follows:

- MR64 has strong security guarantees across all use cases (unlike MRAC and BPSW). These guarantees can be improved by switching to MR128, but we consider the guarantees of MR64 to be sufficient for perhaps all but the most stringent requirements.
- MR64 is easy to implement, while a test like BPSW requires significant additional code. We give a reference implementation of the BPSW test in Appendix B.1 as it could be implemented in OpenSSL 1.1.1c. This helps provide an understanding of the increase in code complexity involved in using this test.
- MR64 with an input-size-dependent choice of r outperforms the current approach used in OpenSSL (MRAC with fixed $r = 2047$) up to $k = 1024$ and remains competitive with MRAC even for larger inputs. (Obviously OpenSSL could also be made faster by tuning r, but this would not improve security for malicious inputs).
- MR64 permits a very simple API, with a single input (the number being tested) and a single output (whether the input was composite or probably prime), whilst still allowing input-size-dependent tuning of r.

Table 5.5 shows our recommended values of r to use with MR64, based on the experimental results obtained above. Further small improvements in performance could be obtained by being more precise in setting r values and by further partitioning the set of k values, but the gains would be marginal.

We further validate this selection of MR64 in the next section, where we examine the performance of different tests when used as part of prime *generation* (as opposed to testing).

k	r
$k \in [1, 512]$	64
$k \in [513, 1024]$	128
$k \in [1025, 2048]$	384
$k \in [2049, 3072]$	768
$k \in [3073, \infty)$	1024

Table 5.5: Recommended values of r for use with the MR64 primality test.

5.4 Prime Generation

In this section, we want to assess the impact of our choice of primality test on a key use case, prime generation. We focus on the scenario where our primality test is used as a drop-in replacement for the existing primality test in OpenSSL, without making any modifications to the prime generation code. We are not suggesting this should be done in practice, but are merely evaluating a strawman example when switching to our proposed test.

5.4.1 Experimental Approach

In order to establish a benchmark, we first use OpenSSL's prime number generating function `BN_generate_prime_ex` as it appears in the standard library. As discussed in detail in Section 5.2.2, this involves sieving with $s = 2047$ primes and using the OpenSSL primality test that consumes t rounds of MR testing on a sequence of candidates n, $n+2$, ..., restarting the procedure from scratch whenever an MR test fails. Here t is determined as in Table 5.1 (i.e. the test is what we call MRAC). Importantly, OpenSSL exploits the rich API of its primality test to switch off trial division in the primality tests, since that trial division is already taken care of by the cheaper sieving step.

Next, we change the underlying primality test to use our selected test: MR64 with input-length-dependent trial division (as per Table 5.5), keeping all other aspects of OpenSSL's prime generation procedure the same. All the trial division done in our underlying primality test is of course redundant, because of the sieving step carried

k	r used	MR64	MRAC	Overhead
512	64	12.37	8.859	40%
1024	128	60.83	45.20	35%
2048	384	385.2	268.5	43%
3072	768	1379	946.7	46%

Table 5.6: Running time (in ms) for prime generation using our proposed primality test (MR64 with input-length-dependent trial division) and current OpenSSL primality test (MRAC with no trial division). Each timing is based on 2^{20} trials.

out in OpenSSL's prime generation code. However, with our deliberately simplified API for primality testing, that extra work would be unavoidable. Similarly, our underlying primality test performs more rounds of MR testing (64 instead of the 3-5 used in MRAC) when a prime is finally encountered. It is the amount of this extra work that we seek to quantify here.

Our experimental results are shown in Table 5.6. It can be seen that the overhead of switching to our primality test in this use case ranges between 35% and 46%. This is a significant cost for this use case, but recall that the gain is a primality test that has strong security guarantees across all use cases, along with a simple and developer-friendly API.

5.4.2 Cost Modelling

We can build simple cost models which illustrate the performance differences we have observed; see also [102] for a similar model. Sieving can be recast as a one-time trial division of the first candidate n with the first s odd primes (OpenSSL uses $s = 2047$), followed by per candidate updating of a table of remainders. We assume the latter can be done essentially for free compared to other operations and ignore its cost henceforth. Then the average cost of prime generation when the underlying primality test uses up to t rounds of MR testing but no trial division, is given by:

$$\left(\sum_{i=1}^{s} C_i\right) + \left(\ln(2^k) \cdot (1 - \sigma_s)/2\right) \cdot C_{MR} + (t-1) \cdot C_{MR}. \tag{5.5}$$

Here the first term comes from sieving. The second term comes from, on average, inspecting $\ln(2^k) \cdot (1 - \sigma_s)/2$ odd, composite candidates in the sieved version of the

list $n, n+2, n+4, \ldots$ before encountering a prime, and doing 1 MR test to reject each composite (recall that, because of sieving, the density of primes in the list $n, n+2, n+4, \ldots$ is boosted by a factor $1/(1-\sigma_s)$; recall also that almost every random composite is rejected with just 1 MR test). The third term comes from doing a further $t-1$ MR tests when a prime is finally found. To model OpenSSL's performance, we would set t according to Table 5.1.

Note that this analysis ignores the fact that OpenSSL aborts and restarts with a fresh, random value whenever an MR test fails; this effect may be significant in practice and we leave a detailed evaluation to future work. Note also that this modelling deficiency does not affect our experimental results reported in the main body, since they were obtained by measuring the running time of the actual OpenSSL code.

It should be evident from expression (5.5) that, as with trial division, working with large s in the initial sieve is not profitable: eventually, the gains made from decreasing the term $1-\sigma_s$ are outweighed by the cost of initial sieving by trial division. Moreover, this model neglects the true cost of updating the table of remainders between candidates. This cost is linear in s (albeit with a small constant) and so heightens the effect. A more detailed model including this cost could of course be developed.

If we now assume that (redundant) trial division with $r \leq s$ primes is also carried out in the underlying primality test, and that the test uses up to t' rounds of MR testing, then the average cost becomes:

$$\left(\sum_{i=1}^{s} C_i\right) + \left(\ln(2^k) \cdot (1-\sigma_s)/2\right) \cdot ((\sum_{i=1}^{r} C_i) + C_{MR}) + (t'-1) \cdot C_{MR} \qquad (5.6)$$

Here, the additional cost compared to (5.5) is precisely that of doing a full set of r trial divisions for each candidate – this cost is always incurred because when $r \leq s$, all the candidates which might fail trial division at some early stage have already failed on sieving. To model the performance of OpenSSL with our chosen primality test, MR64, t' must be set to 64 rather than the values in Table 5.1; the difference means that, when a prime is finally encountered, the cost of testing it will be higher.

The difference in the costs as expressed in (5.5) and (5.6) is given by:

$$\Big(\ln(2^k) \cdot (1 - \sigma_s)/2\Big) \cdot (\sum_{i=1}^{r} C_i) + \delta_t \cdot C_{MR} \tag{5.7}$$

where $\delta_t = t' - t$, depending on k, is the difference in the maximum number of rounds of MR testing carried out in the two cases.

For MR64 and MRAC, and for k of cryptographic size, δ_t ranges between 59 and 61. For our selected primality test, MR64 with input-length-dependent trial division, r in the above expression is also k-dependent, and is set by Table 5.5. The first term in (5.7) accounts for the cost of redundant trial division over the first r primes for $N := \ln(2^k) \cdot (1 - \sigma_s)/2$ different candidates. Here both r and N are in the range of a few hundred. For example, when $k = 1024$ we set $r = 128$, and when $s = 2047$, we have $N \approx 41$. Hence, when $k = 1024$, we do about 5200 redundant trial divisions, compared to an extra $\delta_t = 59$ MR tests. For this k, the extra MR tests are about 8 times more expensive than the redundant trial divisions (roughly 17.5ms versus 2ms based on our experimental timings). This indicates that the redundant trial division contributes much less to the overhead of prime generation than do the extra MR tests that are necessary to make our primality test secure in all use cases.

5.5 Implementation and Integration in OpenSSL

We communicated our findings to the OpenSSL development team, specifically to Kurt Roeckx, one of the OpenSSL core developers. He did his own performance testing, and concluded that our new API and primality test should be deployed in OpenSSL. In personal communication with Roeckx, we were informed that these changes are slated for inclusion in OpenSSL 3.0, which is scheduled for release in Q4 of 2020.

In more detail, the following changes were made:

- Our proposed API is included via a new, external-facing function (see `https://github.com/openssl/openssl/blob/master/crypto/bn/bn_prime.c#L253`):

  ```
  int BN_check_prime(const BIGNUM *p, BN_CTX *ctx, BN_GENCB *cb)
  ```

```
{
return bn_check_prime_int(p, 0, ctx, 1, cb);
}
```

This code wraps the existing "internal" primality testing function `bn_check_prime_int`. Note that the API has 3 parameters, instead of our desired 1: OpenSSL still needs to pass pointers to context and callback objects for programmatic reasons.

- The "internal" primality testing function `bn_check_prime_int` has been updated to do a minimum of 64 rounds of MR testing (and 128 rounds for 2048+ bit inputs). This deviates slightly from our recommendation to always do 64 rounds of testing – it is more conservative. Note that the average case analysis of [41] is no longer used to set the number of rounds of MR testing in the default case. This function also uses a small table to determine how many primes to use in trial division; the numbers are aligned with our recommendations in Table 5.5. Details are in the new function `calc_trial_divisions`.[9]

- The rest of the OpenSSL codebase has been updated to use the new API, except for the prime generation code. That code has also been updated (see `https://github.com/openssl/openssl/blob/master/crypto/bn/bn_prime.c#L123`). It now uses yet a third internal function for its primality testing (see `bn_prime.c#L170`):

  ```
  bn_is_prime_int(ret, checks, ctx, 0, cb);
  ```

  Here, `checks` determines the number of rounds of MR testing done, and is set to either 64 or 128 according to the input size. In the call,"0" indicates that trial division is no longer done. The number of MR rounds here could have been set based on average case performance, as was formerly the case, rather than worst case, but it seems the OpenSSL developers have opted for simplicity over performance. Not doing trial division inside the primality test is appropriate here because the inputs have already been sieved to remove numbers with small prime factors by this point.

- The "old" and complex external-facing APIs in the functions `BN_is_prime_ex` and `BN_is_prime_fasttest_ex` have been marked for deprecation in OpenSSL

[9] See `https://github.com/openssl/openssl/blob/master/crypto/bn/bn_prime.c#L74`.

3.0: they will only be included in a build of the library in case the environmental variable `OPENSSL_NO_DEPRECATED_3_0` is set.[10]

5.6 Conclusions and Future Work

We have proposed a primality test that is both performant and misuse-resistant, in the sense of presenting a simplest-possible interface for developers. The test balances code simplicity, performance, and security guarantees across all use cases. We have not seen a detailed treatment of this fundamental problem in the literature before, despite the by-now classical nature of primality testing as a cryptographic task. Our recommendations – both for the API and for the underlying primality test – have been adopted in full by OpenSSL and are scheduled for inclusion in OpenSSL 3.0, which is expected to be released in Q4 2020.[11]

We have focussed in this work on regular prime generation. Our work could be extended to consider efficiency of safe-prime generation. Special sieving procedures can be used in this case: if one creates a table of values $n \bmod p_i$, then one can also test $2n + 1$ for divisibility by each of the p_i very cheaply; techniques like this were used in [57] in a slightly different context. Further work is also needed to fully assess the impact of the amount of sieving (s) on the performance of prime generation at different input lengths (k). Our work could also be extended to make a systematic study of prime generation code in different cryptographic libraries. For example, we have already noted that the OpenSSL code aborts and restarts whenever a Miller-Rabin test fails; this behaviour leads to sub-optimal performance, and it would be interesting to see how much the code in OpenSSL and in other leading libraries could be improved.

One can view our work as addressing a specific instance of the problem of how to design simple, performant, misuse-resistant APIs for cryptography. In our discussion of related work, we highlighted other work where this problem has also been considered, in symmetric encryption, key exchange, and secure channels. A broader

[10] See `https://www.openssl.org/docs/manmaster/man3/BN\_is\_prime\_fasttest\_ex.html` for details.

[11] See `https://www.openssl.org/blog/blog/2019/11/07/3.0-update/`.

research effort in this direction seems likely to yield significant rewards for the security of cryptographic software. As here, it may occasionally also yield improved performance.

Chapter 6

Conclusion

In this chapter we provide an overall summary for this thesis, and then go on to give a more detailed breakdown with respect to each individual chapter. We also briefly mention avenues for future work.

In this thesis we have provided a holistic analysis of primality testing and its use as a mathematical tool within cryptography. Using primality testing as a case study, we journeyed from the very inception of the fundamental primality testing algorithms, to their standardisation and recommended configuration, to the real world implementation. Using this systematic style of analysis, we are able to uncover flaws in these primality tests at many different stages. Some flaws are inherent to the design of the primality test, for example the nature of pseudoprimes for the Miller-Rabin test. In this case, we expanded existing techniques for creating such pseudoprimes, and furthered their use as malicious parameter sets for public-key cryptography. We also uncovered numerous flaws in the implementation of primality testing across cryptographic libraries. The fault here is usually contingent upon the authors not fully understanding the importance of selecting random bases when performing Miller-Rabin, or not performing enough rounds of testing. We also witnessed a common misunderstanding in distinguishing between when the primality test is performed on either random, or adversarial input – this often resulted in a failure to use the correct error bounds or the correct context for applying these bounds to the result of the test. As an outcome of this analysis, we were able to impact the primality testing procedures in numerous implementations across different libraries to provide security against these pseudoprimes in the real world. For example, we worked with the developers of the most widely used cryptographic library, OpenSSL, at many different stages of the primality testing functionality. OpenSSL

have fully adopted the solutions provided by this work, from amending the public key parameter checking function for Diffie-Hellman, working to make the documentation clearer on testing random or malicious input, to even re-writing the core primality tests to provide a more robust API. We also worked with other vendors such as Apple, Bouncy Castle, Botan, Mozilla and WolfSSL to improve the security of their primality testing procedures.

In **Chapter 3** we explored primality testing in the adversarial setting and its impact on Diffie-Hellman parameter testing. Our main finding is that leading libraries are not designed for this setting, and are therefore often vulnerable to accepting maliciously chosen composite inputs – as being prime. We can generally classify the underlying cause of the failure in prime classification accuracy as a non-consideration of the adversarial setting. More explicitly, we can categorise most failures in terms of how the bases for Miller-Rabin are chosen, i.e. fixed base, predictable bases, insufficient number of bases. Apple's corecrypto and CommonCrypto, Mini-GMP, JSBN, Cryptlib, LibTomMath, LibTomCrypt and WolfSSL all fail due to the selection of bases from a fixed list, whereas Mozilla's NSS, GNU GMP, and GoLang pre-1.8 all suffer from predictable bases. OpenSSL, Libgcrypt, Botan and Bouncy Castle C# all have options to run as many rounds of Miller-Rabin as the user desires, but either default to, or call the test (elsewhere in the library) with too few rounds. Problems left open by this work are to find malicious Diffie-Hellman parameter sets that can fool primality tests in the safe-prime setting (this is addressed later in Chaper 4). Further work may also include an investigation into which other seemingly innocuous assumptions concerning domain parameters in the literature can be undermined in a similar fashion.

In **Chapter 4** we considered the problem left open from Chapter 3 of constructing Diffie-Hellman parameters which pass parameter validation functions that test for safe primes, but for which the Discrete Logarithm Problem is relatively easy to solve. We then went on to provide malicious parameter sets for elliptic curve Diffie-Hellman, in a method analogous to the finite field case. This chapter dived much deeper into the existence of pseudoprimes for the Miller-Rabin test – to determine if there are other forms of composite numbers that possess a significant probability (perhaps not optimal) to be declared as prime. The Monier-Rabin bound is synonymous with understanding the accuracy of the Miller-Rabin test; it states that

any odd composite $n \neq 9$ can have at most $\varphi(n)/4$ non-witnesses, and therefore can pass a single round of Miller-Rabin with probability at most $\approx 1/4$. We were successful in expanding the bound, by proving that any odd composite number n with m distinct prime factors can have at most $\varphi(n)/2^{m-1}$ non-witnesses (and therefore a probability of at most $\approx 1/2^{m-1}$ of being declared prime by a single round of Miller-Rabin). We then went on to prove that for any $m \geq 3$ ($m = 2$ was covered significantly in Chapter 3), this maximum bound of non-witnesses can be achieved, and happens if and only if n is a Carmichael number with each factor congruent to 3 mod 4. Not only does this vastly increase the knowledge of how pseudoprimes for the Miller-Rabin test are distributed among the integers, it also gives us the ability to create relatively smooth numbers that still posses a significant probability of falsely being declared prime. This was instrumental in being able to create pseudoprimes to be part of malicious parameter sets (for both finite field and elliptic curve Diffie-Hellman) that allow the Discrete Logarithm Problem to be relatively easy to solve. In the remainder of this chapter, we harnessed techniques from Erdös [51] and Granville and Pomerance [67] to a give methodology for creating such pseudoprimes in an efficient manner. Further work in this area might include additional analysis on the sieving techniques for generation of primes (particularly the ones that form the factors of Carmichael numbers), so that large pseudoprimes with many more factors can be more efficiently generated.

In **Chapter 5** we focused on the development of an API for primality testing that stands robust against misuse within the context of testing input from any source. This is an effort to eliminate the common pitfall identified in Chapters 3 and 4: a failure to consider the case of testing on adversarial input. We provided various different options for the core testing procedure within a primality test, and compared their performance (both accuracy and efficiency) against each other, along with the current implementation in OpenSSL. This work also allowed us to perform a more detailed analysis of a competitor to the Miller-Rabin test: the Baillie-PSW test. The Baillie-PSW test is becoming increasingly more popular within primality testing techniques found in cryptographic libraries and mathematical software, yet there are still serious questions to ask about its performance – both in terms of accuracy and efficiency. In this chapter, we focused on giving a more concrete analysis of the efficiency of the test, particularly with respect to how long the Miller-Rabin test takes to test the same input. We studied the tests in multiple contexts, from testing

random input, to maliciously generated input, to how the time taken to generate prime numbers is affected when this was the core primality test used. This provided us with the ability to propose a primality test that is both performant and misuse-resistant, in the sense of presenting a simplest-possible interface for developers. Our recommendations – both for the API and for the underlying primality test – have been adopted in full by OpenSSL and are scheduled for inclusion in OpenSSL 3.0, which is expected to be released in Q4 2020. Further work in this area would include the continued study of the Baillie-PSW test, in particular the search for pseudoprimes for the test. Our work could also be extended to make a systematic study of prime generation code in different cryptographic libraries. For example, we have already noted that the OpenSSL code aborts and restarts whenever a Miller-Rabin test fails; this behaviour leads to sub-optimal performance, and it would be interesting to see how much the code in OpenSSL and in other leading libraries could be improved. A broader research effort could also be made within the direction of how to design simple, performant, misuse-resistant APIs for cryptography.

Appendix A

Implementation Code

Here we provide three example implementations of some of the methods described in Chapter 4. Namely, a SAGE code implementation of the Erdös Method for generating Carmichael numbers, the C code used to generate a Carmichael number q with 9 prime factors such that $p = 2q + 1$ is a 1024 bit prime, and the SAGE code for the first step of the algorithm of Bröker and Stevenhagen in the case where N, the target group order, has 3 prime factors.

A.1 SAGE code of the Erdös Method for Generating Carmichael Numbers

We present below our SAGE code implementation of the Erdös Method for generating Carmichael numbers. This particular code was used to generate the Carmichael numbers with 8 and 16 factors in Example 4.3.

```
import itertools
from operator import mul
from sage.arith.functions import LCM_list


def all_combinations(any_list):
    """
    Wrapper for itertools to generate all possible combinations of all
    (non trivial) sizes.
    """
    return itertools.chain.from_iterable(
        itertools.combinations(any_list, i + 1)
        for i in xrange(len(any_list)))


def LCMpim1(n):
    """
    Takes as input n: a list of integers p_i and returns the lcm(p_i-1) for all i
    """
    pim1list = []
```

A.1 SAGE code of the Erdös Method for Generating Carmichael Numbers

```
    for pi in n:
        pim1 = pi - 1
        pim1list.append(pim1)
    return LCM_list(pim1list)


def listbuild(L):
    """
    Takes as input a (highly composite) number L and returns a list of all primes
    p such that p-1 | L where p does not divide L. We include the additional
    requirement that p = 3 mod 4.
    """
    a = list(factor(L))
    p = []
    for y in a:
        for i in range(0, y[1]):
            p.append(y[0])

    pvals = all_combinations(p)
    ps = []
    for pp in pvals:
        t = reduce(mul, pp, 1)
        tt = t + 1
        if tt.is_prime(proof=False) and L % tt != 0:
            if tt not in ps:
                ps.append(tt)
    pps = []
    ps.sort()
    # we now filter results to only inlude p with p = 3 mod 4
    for p in ps:
        if p % 4 == 3:
            pps.append(p)
    return pps


def erdos_build(factors, L, k):
    """
    This function takes a list of possible factors, a (highly composite) integer L
    and k, and produces a Carmichael number with k factors sampled from "factors"
    such that the LCM of each factor p_i - 1 is equal to L. Output is parsed as
    n,[p_1,p_2,...,p_k] where n = p_1 * p_2 * ... * p_k.
    """
    if k <=2:
        print "Choice of factors must be >=3"
        return 0
    for i in itertools.combinations(factors, k):
        v = reduce(mul, i, 1)
        if v % L == 1:
            fin = list(i)
            fin.sort()
            if LCMpim1(fin) == L:
                return [v,fin]
    print "None found, try increasing size of factor list"


L = 53603550
factors = listbuild(L)
print factors, L, len(factors)

print erdos_build(factors, L, 8)
print erdos_build(factors, L, 16)
```

A.2 C Code of the Modified Granville and Pomerance Method for Generating Carmichael Numbers q such that $p = 2q + 1$ is Prime

We present below our C code used to generate a Carmichael number q with 9 prime factors such that $p = 2q + 1$ is a 1024 bit prime as in Example 4.5.

```
#define _XOPEN_SOURCE 500
#include <stdint.h>
#include <stdio.h>                          /* printf() */
#include <stdlib.h>     /* abort() */
#include <unistd.h>     /* getopt() */
#include <gmp.h>

/* Command Line Parsing  */
#define DEFAULT_COUNT 37
#define DEFAULT_OFFSET 0

struct _cmdline_params_struct {
  uint32_t count;  // we use this for parallelisation
  uint32_t offset; //< how much we want to offset the starting value of k by
};

typedef struct _cmdline_params_struct cmdline_params_t[1];

static inline void print_help_and_exit() {
  printf("-c    log2 of number of trials (default: %d)\n", DEFAULT_COUNT);
  printf("-o    offset on starting k value, where offset*c (default: %d)\n",
      DEFAULT_OFFSET);
  abort();
}

static inline void parse_cmdline(cmdline_params_t params, int argc,
    char *argv[]) {
  params->count = DEFAULT_COUNT;
  params->offset = DEFAULT_OFFSET;

  int c;
  while ((c = getopt(argc, argv, "c:o:")) != -1) {
    switch(c) {
    case 'c':
      params->count = (uint32_t)strtoul(optarg, NULL, 10);
      break;
    case 'o':
      params->offset = (int32_t)strtoul(optarg,NULL, 10);
      break;
    case ':':  /* without operand */
      print_help_and_exit();
    case '?':
      print_help_and_exit();
    }
  }
  printf("-c %d -o %d\n",
       params->count, params->offset);
}

/* Logging */
void logit(mpz_t q, mpz_t q1, mpz_t q2, mpz_t q3, mpz_t q4, mpz_t q5,
   mpz_t q6, mpz_t q7,mpz_t q8, mpz_t q9) {

  char tmp[2000];
```

```
  snprintf(tmp, 2000, "0x%s:0x%s:0x%s:0x%s:0x%s:0x%s:0x%s:0x%s:0x%s:0x%s",
   mpz_get_str(NULL, 16, q), mpz_get_str(NULL, 16, q1), mpz_get_str(NULL, 16, q2),
   mpz_get_str(NULL, 16, q3), mpz_get_str(NULL, 16, q4), mpz_get_str(NULL, 16, q5),
   mpz_get_str(NULL, 16, q6), mpz_get_str(NULL, 16, q7), mpz_get_str(NULL, 16, q8),
   mpz_get_str(NULL, 16, q9));
  FILE *fh = fopen("CARM-9.log", "a");
  fprintf(fh, "%s\n", tmp);
  fclose(fh);
}

/*
 * Function:  main
 * --------------------
 * This function uses the modified Granville Pomerance method to generate a
 * Carmichael number q of cryptographic size, such that N = 2q+1 is prime.
 *
 * This function is currently not set up for generality, and does not perform
 * sanity checks. We specifically set up an instance of this code to search for
 * a single valid example. This is the 9 factored example that is given a starting
 * Carmichael number p = p_1 *...* p_9 generated previously by the Erdos method.
 *
 * The function iterates through kprime (k') values to construct:
 *    m = kL + 1, where k = k' * s
 *    then q_i = M(p_i-1)+1 for all i
 * such that q = q_1 * ... * q_9 is approx 1023 bits.
 *
 * We then test each q_i for primality, iterating to the next k' value if composite.
 * Finally, if all q_i are prime, we construct q = q_1 * ... * q_9 and test if
 * N = 2q+1 is prime. If true, we log q, and its factors.
 *
 */
int main(int argc, char *argv[])
{
  mpz_t s,p1,p2,p3,p4,p5,p6,p7,p8,p9,q,q1,q2,q3,q4,q5,q6,q7,q8,q9,kprime,fudge2,
          fudge3,fudge4,fudge5,k,m,off,L,N;
  mpz_init(q);
  mpz_init(q1);
  mpz_init(q2);
  mpz_init(q3);
  mpz_init(q4);
  mpz_init(q5);
  mpz_init(q6);
  mpz_init(q7);
  mpz_init(q8);
  mpz_init(q9);
  mpz_init(k);
  mpz_init(m);
  mpz_init(off);
  mpz_init(N);
  int res;

  cmdline_params_t params;
  parse_cmdline(params, argc, argv);
  // here we set up our specific starting Carmichael number p and other parameters
  mpz_init_set_str(s, "3", 10);
  mpz_init_set_str(kprime, "1", 10);
  mpz_init_set_str(fudge2, "1", 10);
  mpz_init_set_str(fudge3, "1", 10);
  mpz_init_set_str(fudge4, "1", 10);
  mpz_init_set_str(fudge5, "1", 10);
  mpz_init_set_str(p1, "70", 10);
  mpz_init_set_str(p2, "130", 10);
  mpz_init_set_str(p3, "646", 10);
  mpz_init_set_str(p4, "1870", 10);
  mpz_init_set_str(p5, "4522", 10);
  mpz_init_set_str(p6, "4750", 10);
```

A.2 C Code of the Modified Granville and Pomerance Method for Generating Carmichael Numbers q such that $p = 2q + 1$ is Prime

```
mpz_init_set_str(p7, "46750", 10);
mpz_init_set_str(p8, "432250", 10);
mpz_init_set_str(p9, "350350", 10);
mpz_init_set_str(L, "565815250", 10);
uint64_t Lbits = 30;

mpz_init_set_ui(off,params->offset);
mpz_mul_2exp(off,off,params->count);

size_t p1bits = mpz_sizeinbase (p1, 2);
size_t sbits = mpz_sizeinbase (s, 2);

// we now make some speicific alterations to ensure the final N is 1024 bits
uint64_t t = 9;
uint64_t fudgefactor = 1;
uint64_t power = 113 - (t/2 -1) -p1bits - Lbits - sbits + fudgefactor;

mpz_mul_2exp(kprime,kprime,power);
mpz_mul_2exp(fudge2,fudge2,power-1);
mpz_mul_2exp(fudge3,fudge3,power-4);
mpz_mul_2exp(fudge4,fudge4,power-5);
mpz_mul_2exp(fudge5,fudge5,power-6);
mpz_sub(kprime,kprime,fudge2);
mpz_sub(kprime,kprime,fudge3);
mpz_sub(kprime,kprime,fudge4);

if (params->offset != 0) {
  mpz_add(kprime,kprime,off);
}
// The following for loop accounts for the bulk of the time to run
for (uint64_t i = 0; i <= (1ULL)<<params->count; i++){
  mpz_add_ui(kprime,kprime,1);
  mpz_mul(k,kprime,s);
  mpz_mul(m,k,L);
  mpz_add_ui(m,m,1);

  //q1
  mpz_mul(q1,m,p1);
  mpz_add_ui(q1,q1,1);
  res= mpz_probab_prime_p (q1, 2);
  if (!res) {
      continue;
  }
  //q2
  mpz_mul(q2,m,p2);
  mpz_add_ui(q2,q2,1);
  res= mpz_probab_prime_p (q2, 2);
  if (!res) {
      continue;
  }
  //q3
  mpz_mul(q3,m,p3);
  mpz_add_ui(q3,q3,1);
  res= mpz_probab_prime_p (q3, 2);
  if (!res) {
      continue;
  }
  //q4
  mpz_mul(q4,m,p4);
  mpz_add_ui(q4,q4,1);
  res= mpz_probab_prime_p (q4, 2);
  if (!res) {
      continue;
  }
  //q5
  mpz_mul(q5,m,p5);
```

```
    mpz_add_ui(q5,q5,1);
    res= mpz_probab_prime_p (q5, 2);
    if (!res) {
        continue;
    }
    //q6
    mpz_mul(q6,m,p6);
    mpz_add_ui(q6,q6,1);
    res= mpz_probab_prime_p (q6, 2);
    if (!res) {
        continue;
    }
    //q7
    mpz_mul(q7,m,p7);
    mpz_add_ui(q7,q7,1);
    res= mpz_probab_prime_p (q7, 2);
    if (!res) {
        continue;
    }
    //q8
    mpz_mul(q8,m,p8);
    mpz_add_ui(q8,q8,1);
    res= mpz_probab_prime_p (q8, 2);
    if (!res) {
        continue;
    }
    //q9
    mpz_mul(q9,m,p9);
    mpz_add_ui(q9,q9,1);
    res= mpz_probab_prime_p (q9, 2);
    if (!res) {
        continue;
    }

    mpz_mul(q,q1,q2);
    mpz_mul(q,q,q3);
    mpz_mul(q,q,q4);
    mpz_mul(q,q,q5);
    mpz_mul(q,q,q6);
    mpz_mul(q,q,q7);
    mpz_mul(q,q,q8);
    mpz_mul(q,q,q9);

    mpz_mul_2exp(N,q,1);
    mpz_add_ui(N,N,1);
    res= mpz_probab_prime_p (N, 2);
    if (!res) {
      continue;
    }
    printf("PRIME!\n" );
    logit(q,q1,q2,q3,q4,q5,q6,q7,q8,q9);
}

mpz_clear(s);
mpz_clear(p1);
mpz_clear(p2);
mpz_clear(p3);
mpz_clear(p4);
mpz_clear(p5);
mpz_clear(p6);
mpz_clear(p7);
mpz_clear(p8);
mpz_clear(p9);
mpz_clear(q1);
mpz_clear(q2);
mpz_clear(q3);
```

```
  mpz_clear(q4);
  mpz_clear(q5);
  mpz_clear(q6);
  mpz_clear(q7);
  mpz_clear(q8);
  mpz_clear(q9);
  mpz_clear(kprime);
  mpz_clear(fudge2);
  mpz_clear(fudge3);
  mpz_clear(fudge4);
  mpz_clear(fudge5);
  mpz_clear(k);
  mpz_clear(m);
  mpz_clear(off);
  mpz_clear(L);
  mpz_clear(N);
  return 0;
}
```

A.3 SAGE code for Algorithm of Bröker and Stevenhagen

We present below our SAGE code for the first step of the algorithm of Bröker and Stevenhagen in the case where N, the target group order, has 3 prime factors. This code was written by Steven Galbraith, with the exception of the parameters p, q, r which were from myself.

```
# Generate elliptic curve using CM with group order divisible by product p*q*r
# that is a fake prime.


# Cornacchia algorithm
def Cornacchia( A, B, D ):
   a = A
   b = B
   while (b^2 > A):
     rrem = int( Mod(a,b) )
     a = b
     b = rrem
   x = b
   f2 = (A - x^2) / -D
   f = int( sqrt( f2 ))
   return x, f



# [584170554761513436280134435700062590076357016263612392265089290457585365018 51,
p = 120969320416809549586937 71
q = 362907961250428648760813 11
r = 133066252458490504545631471
N = p*q*r


DBOUND = -2000;

# First try to construct a curve with N points
D = -3
```

```
while (D > DBOUND):
  if (1 == legendre_symbol( D, p )) and (1 == legendre_symbol( D, q )) and
      (1 == legendre_symbol( D, r )):
    F = GF( p )
    x01 = int( sqrt( F( D ) ))
    F = GF( q )
    x02 = int( sqrt( F( D ) ))
    F = GF( r )
    x03 = int( sqrt( F( D ) ))
# There are 8 possible choices for x0 coming from the 2^3 choices of sign
#  +/- x01, +/- x02, +/- x03
    ct = 0
    while (ct < 8):
      x0 = crt( crt( x01, x02, p, q ), x03, p*q, r )
      while (0 != Mod(x0^2-D,4*N)):
        x0 = x0+N
      x, f = Cornacchia( 4*N, x0, D )
      if ( 0 == (x^2 - D*f^2 - 4*N)):
         pp = int( N + x + 1 )
         if is_prime(pp):
            print "Success (D,x,f) = ", D, x, f
            print "And get a prime p = ", pp
         pp = int( N - x + 1 )
         if is_prime(pp):
            print "Success with other sign (D,x,f) = ", D, x, f
            print "And get a prime p = ", pp
      x01 = p - x01
      if (0 == (ct % 2)):
        x02 = q - x02
      if (0 == (ct % 4)):
        x03 = r - x03
      ct = ct + 1
  D = D - 4


# Now consider curves whose number of points is a multiple of 2*N
# Algorithm is basically the same except D now must be even
c = 1
while (c < 5):
  NN = 2*c*N
  c = c + 1
  D = -4
  while (D > DBOUND):
    D = D - 4
    DD = D
    if (1 == legendre_symbol( D, p )) and (1 == legendre_symbol( D, q )) and
        (1 == legendre_symbol( D, r )):
      F = GF( p )
      x01 = int( sqrt( F( DD ) ))
      F = GF( q )
      x02 = int( sqrt( F( DD ) ))
      F = GF( r )
      x03 = int( sqrt( F( DD ) ))
      ct = 0
      while (ct < 8):
        x0 = crt( crt( x01, x02, p, q ), x03, p*q, r )
        chk=0
        while (0 != Mod(x0^2-DD,4*NN)) and (chk < 100):
          chk = chk+1
          x0 = x0+N
        x, f = Cornacchia( 4*NN, x0, D )
        if ( 0 == (x^2 - DD*f^2 - 4*NN)):
           pp = int( NN + x + 1 )
           if is_prime(pp):
              print "Success (D,x,f) = ", DD, x, f
              print "And get a prime p = ", pp
```

```
    pp = int( NN - x + 1 )
    if is_prime(pp):
       print "Success with other sign (D,x,f) = ", DD, x, f
       print "And get a prime p = ", pp
x01 = p - x01
if (0 == (ct % 2)):
  x02 = q - x02
if (0 == (ct % 4)):
  x03 = r - x03
ct = ct + 1
```

Appendix B

Baillie-PSW Test

B.1 Reference Implementation of the Baillie-PSW Test

For completeness, we include here our code that implements a Baillie-PSW primality test in the context of OpenSSL's `bn_prime.c` from version 1.1.1c. `bn_prime_bpsw.c`

```
/*
 * Copyright 1995-2018 The OpenSSL Project Authors. All Rights Reserved.
 *
 * Licensed under the OpenSSL license (the "License").  You may not use
 * this file except in compliance with the License.  You can obtain a copy
 * in the file LICENSE in the source distribution or at
 * https://www.openssl.org/source/license.html
 */

#include <stdio.h>
#include <time.h>
#include "internal/cryptlib.h"
#include "bn_lcl.h"
#include "bn_prime.h"

static int witness(BIGNUM *w, const BIGNUM *a, const BIGNUM *a1,
                   const BIGNUM *a1_odd, int k, BN_CTX *ctx,
                   BN_MONT_CTX *mont);
static int probable_prime(BIGNUM *rnd, int bits, prime_t *mods);
static int probable_prime_dh_safe(BIGNUM *rnd, int bits,
                                  const BIGNUM *add, const BIGNUM *rem,
                                  BN_CTX *ctx);
static int BN_lucas_test_ex(BIGNUM * n);
static int BN_jacobi(BIGNUM *a, BIGNUM *n);
static BIGNUM * BN_lucas_sequence(BIGNUM *d, BIGNUM *k, BIGNUM *n);
static BIGNUM * BN_is_perfect_square(BIGNUM * C);
static int BN_is_prime_BPSW_ex(BIGNUM *a, BN_CTX *ctx_passed,
                               int do_trial_division, BN_GENCB *cb);


int BN_GENCB_call(BN_GENCB *cb, int a, int b)
{
    /* No callback means continue */
    if (!cb)
        return 1;
    switch (cb->ver) {
    case 1:
        /* Deprecated-style callbacks */
```

```
        if (!cb->cb.cb_1)
            return 1;
        cb->cb.cb_1(a, b, cb->arg);
        return 1;
    case 2:
        /* New-style callbacks */
        return cb->cb.cb_2(a, b, cb);
    default:
        break;
    }
    /* Unrecognised callback type */
    return 0;
}

int BN_is_prime_BPSW_ex(BIGNUM *a, BN_CTX *ctx_passed,
                            int do_trial_division, BN_GENCB *cb)
{
    int i, j, l, ret = -1;
    int k;
    BN_CTX *ctx = NULL;
    BIGNUM *A1, *A1_odd, *check = BN_new(); /* taken from ctx */
    BN_MONT_CTX *mont = NULL;

    BN_set_word(check, 2); //only testing MR to base 2

    /* Take care of the really small primes 2 & 3 */
    if (BN_is_word(a, 2) || BN_is_word(a, 3))
        return 1;

    /* Check odd and bigger than 1 */
    if (!BN_is_odd(a) || BN_cmp(a, BN_value_one()) <= 0)
        return 0;

    /* first look for small factors */
    if (do_trial_division) {
        for (i = 1; i < TRIAL_DIVISION_PRIMES; i++) {
            BN_ULONG mod = BN_mod_word(a, primes[i]);
            if (mod == (BN_ULONG)-1)
                goto err;
            if (mod == 0)
                return BN_is_word(a, primes[i]);
        }
        if (!BN_GENCB_call(cb, 1, -1))
            goto err;
    }

    if (ctx_passed != NULL)
        ctx = ctx_passed;
    else if ((ctx = BN_CTX_new()) == NULL)
        goto err;
    BN_CTX_start(ctx);

    A1 = BN_CTX_get(ctx);
    A1_odd = BN_CTX_get(ctx);

    if (check == NULL)
        goto err;

    /* compute A1 := a - 1 */
    if (!BN_copy(A1, a) || !BN_sub_word(A1, 1))
        goto err;

    /* write  A1  as  A1_odd * 2^k */
    k = 1;
    while (!BN_is_bit_set(A1, k))
        k++;
```

```
    if (!BN_rshift(A1_odd, A1, k))
        goto err;

    /* Montgomery setup for computations mod a */
    mont = BN_MONT_CTX_new();
    if (mont == NULL)
        goto err;
    if (!BN_MONT_CTX_set(mont, a, ctx))
        goto err;

    j = witness(check, a, A1, A1_odd, k, ctx, mont);
    if (j == -1)
        goto err;
    if (j) {
        ret = 0;
        goto err;
    }
    if (!BN_GENCB_call(cb, 1, i))
        goto err;

    ret = 1;

    l = BN_lucas_test_ex(a);
    if (!l) {
        ret = 0;
        goto err;
      }

 err:
    if (ctx != NULL) {
        BN_CTX_end(ctx);
        if (ctx_passed == NULL)
            BN_CTX_free(ctx);
    }
    BN_MONT_CTX_free(mont);

    return ret;
}

int BN_lucas_test_ex(BIGNUM * n){
  //performs a Lucas test (with Selfridge's paramters) on n
  BIGNUM *two = BN_new();
  BN_set_word(two, 2);

  // sanity check input, n odd and > 2
  if (BN_cmp(two,n)==1) { // 1 if a > b i.e b < a
    BN_free(two);
    return 0;
  }
  if (BN_cmp(n,two)==0) {
    BN_free(two);
    return 1;
  }
  if (!BN_is_odd(n)) {
    BN_free(two);
    return 0;
  }

  BN_CTX *ctx = BN_CTX_new();
  BIGNUM *result = BN_new();
  BIGNUM *zero= BN_new();
  BIGNUM *np1 = BN_new();
  BIGNUM *minusone = BN_new();
  BIGNUM *u = BN_new();
  BIGNUM *d = BN_new();
  BIGNUM *minusnineteen = BN_new();
```

```
  int32_t J;
  int32_t res;

  const char *m1 = "-1";
  const char *m19 = "-19";
  BN_add(np1,n,BN_value_one());
  BN_zero(zero);
  BN_dec2bn(&minusone, m1);
  BN_dec2bn(&minusnineteen, m19);
  BN_set_word(d, 5);

 // while jacobi(d,n) != -1
  while ((J = BN_jacobi(d,n))!= -1) {
    if (J==0) { // if jacobi(d,n) == 0 then d | n, i.e n is composite
      res = 0;
      goto free;
    }
    if (BN_cmp(zero,d)==1) { // 0>d
      BN_mul(d,d,minusone,ctx);
      BN_add(d,d,two);
    }
    else{
      BN_add(d,d,two);
      BN_mul(d,d,minusone,ctx);
    }

    if (BN_cmp(d,minusnineteen)==0 && !(BN_cmp(BN_is_perfect_square(n),zero)==0)) {
      res = 0;
      goto free;
    }
  }
  u = BN_lucas_sequence(d,np1,n);
  BN_mod(result,u,n,ctx);
  if (BN_cmp(result,zero)==0) {
    res = 1;
    goto free;
  }
  else{
    res = 0;
    goto free;
  }

  free:
    BN_CTX_free(ctx);
    BN_free(result);
    BN_free(zero);
    BN_free(np1);
    BN_free(minusone);
    BN_free(two);
    BN_free(u);
    BN_free(d);
    return res;
}

int BN_jacobi(BIGNUM *a, BIGNUM *n){
  // computes jacobi symbol of (a/n), currently returns 2 if a,n are invalid input
  BIGNUM *x = BN_new();
  BIGNUM *y = BN_new();
  BIGNUM *halfy = BN_new();
  BIGNUM *r = BN_new();
  BIGNUM *s = BN_new();

  BN_CTX *ctx = BN_CTX_new();
  BN_nnmod(x,a,n,ctx);
  BN_copy(y,n);
```

```
  int J  = 1;
  int k = 0;

  BIGNUM *three = BN_new();
  BN_set_word(three, 3);
  BIGNUM *four = BN_new();
  BN_set_word(four, 4);
  BIGNUM *five = BN_new();
  BN_set_word(five, 5);
  BIGNUM *eight = BN_new();
  BN_set_word(eight, 8);

  if (!BN_is_odd(n)||BN_cmp(n,BN_value_one()) <= 0) {
    J = 2;
    goto free;
  }

  while (BN_cmp(y,BN_value_one()) == 1) { // while y > 1
    BN_mod(x,x,y,ctx);
    BN_rshift1(halfy,y);
    if (BN_cmp(x,halfy)==1) {
      BN_sub(x,y,x);
      BN_mod(r,y,four,ctx);
      if (BN_cmp(r,three)==0) {
        J = -J;
      }
    }
    if (BN_is_zero(x)) {
      //gcd(a,n)!=1 so we return 0
      J = 0;
      goto free;
    }
    //count the zero bits in x, i.e the largest value of n s.t 2^n divides x evenly.
    k = 0;
    while (!BN_is_bit_set(x, k)) {
      k++;
    }
    BN_rshift(x,x,k);

    if (k%2) {
      BN_mod(s,y,eight,ctx);
      if (BN_cmp(s,three)==0 || BN_cmp(s,five)==0) {
        J = -J;
      }
    }
    BN_mod(r,x,four,ctx);
    BN_mod(s,y,four,ctx);
    if (BN_cmp(r,three)==0 && BN_cmp(s,three)==0) {
      J = -J;
    }
    BN_swap(x,y);
  }

  free:
     BN_CTX_free(ctx);
     BN_free(x);
     BN_free(y);
     BN_free(halfy);
     BN_free(r);
     BN_free(s);
     BN_free(three);
     BN_free(four);
     BN_free(five);
     BN_free(eight);
     return J;
}
```

```
void BN_rshift1_round(BIGNUM *r, BIGNUM *a){
  // temporary fix as part of code demo, but the rounding in BN_rshift1
  // is not consistant with python/java across positive and negative numbers.
  // This function adds one before the shift if a is negative and performs
  // BN_rshift1 normally otherwise. e.g this function rounds -127/2 = -63.5
  // to -64 (toward -infinity), where as BN_rshift1 would round to -63 (toward 0)
  // This is needed in my implementation of jacobi symbol calculation.
  //Can't simply negate result, as we still want 127/2 = 63.

  BIGNUM *zero= BN_new();
  BIGNUM *one= BN_new();
  BN_zero(zero);
  BN_one(one);

  if (BN_cmp(zero,a)==1) { //a < 0
    BN_sub(r,a,one);
    BN_rshift1(r,r);
  }
  else{
    BN_rshift1(r,a);
  }
  BN_free(zero);
  BN_free(one);
}

BIGNUM * BN_lucas_sequence(BIGNUM *d, BIGNUM *k, BIGNUM *n){
   //computes the Lucas sequence U_k modulo n, where d = p^2 -4q
   BN_CTX *ctx = BN_CTX_new();
   BIGNUM *kp1 = BN_new();
   BIGNUM *u = BN_new();
   BIGNUM *v = BN_new();
   BIGNUM *u2 = BN_new();
   BIGNUM *v2 = BN_new();
   BIGNUM *r= BN_new();
   BIGNUM *zero= BN_new();
   BIGNUM *one= BN_new();

   BN_add(kp1,k,BN_value_one());
   BN_zero(zero);
   BN_one(one);
   BN_one(u);
   BN_one(v);

   size_t k_bits = BN_num_bits(kp1) -1;

   for (size_t i = k_bits-1; i != (size_t) -1; --i) {
     BN_mod_mul(u2,u,v,n,ctx);
     BN_mod_sqr(r,u,n,ctx); //r = u^2 mod n
     BN_mod_mul(r,r,d,n,ctx); // r = r *d  = u^2 *d (mod n)
     BN_mod_sqr(v2,v,n,ctx); //v2 = v^2 mod n
     BN_mod_add(v2,v2,r,n,ctx); // v2 = v2 + r = v^2 + (u^2*d) (mod n)

     if (BN_is_odd(v2)) {
       BN_sub(v2,v2,n);// v2 = v2 - n
     }

     BN_rshift1_round(v2,v2);
     BN_copy(u,u2);
     BN_copy(v,v2);

     if (BN_is_bit_set(k,i)) {
       BN_nnmod(r,v,n,ctx); //r= v mod
       BN_add(u2,u,r); // u2 = u + v mod n

       if (BN_is_odd(u2)) {
```

```
        BN_sub(u2,u2,n);
      }

      BN_rshift1_round(u2,u2);
      BN_mod_mul(r,d,u,n,ctx); // r = d*u mod n
      BN_add(v2,v,r); // v2 = r + v = v + d*u mod n

      if (BN_is_odd(v2)) {
        BN_sub(v2,v2,n);
      }

      BN_rshift1_round(v2,v2);
      BN_copy(u,u2);
      BN_copy(v,v2);
      }
   }
BN_CTX_free(ctx);
BN_free(kp1);
BN_free(v);
BN_free(u2);
BN_free(v2);
BN_free(r);
BN_free(zero);
BN_free(one);
return u;
}

BIGNUM * BN_is_perfect_square(BIGNUM * C){
  //https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf sec C.4
  // checks if C is a perfect square. If so, function returns X where C = X^2
  // else function returns 0
  BIGNUM *one= BN_new();
  BIGNUM *zero= BN_new();
  BIGNUM *ret= BN_new();
  BN_one(one);
  BN_zero(zero);

  if (BN_cmp(one,C)==1) {
    printf("is_perfect_square requires C >=1 \n");
    BN_free(one);
    return zero;
  }
  if (BN_cmp(one,C)==0) {
    BN_free(zero);
    return one;
  }

  BN_CTX *ctx = BN_CTX_new();
  BIGNUM *B = BN_new();
  BIGNUM *X = BN_new();
  BIGNUM *r = BN_new();
  BIGNUM *s = BN_new();
  BIGNUM *X2 = BN_new();
  BIGNUM *two= BN_new();
  size_t c_bits = BN_num_bits(C);
  size_t m = (c_bits+1)/2;

  BN_set_word(two, 2);
  BN_set_bit(B,m);
  BN_add(B,B,C);
  BN_set_bit(X,m);
  BN_sub(X,X,one);
  BN_mul(X2,X,X,ctx);

  for (;;) {
    BN_add(r,X2,C);
```

```
    BN_mul(s,X,two,ctx);
    BN_div(X,NULL,r,s,ctx);
    BN_mul(X2,X,X,ctx);
    if (BN_cmp(B,X2)==1) {
      break;
    }
  }
  if (BN_cmp(X2,C)==0) {
    ret = X;
    goto free;
  }
  else {
    ret = zero;
    goto free;
  }
  free:
    BN_CTX_free(ctx);
    BN_free(B);
    BN_free(r);
    BN_free(s);
    BN_free(X2);
    BN_free(one);
    BN_free(two);
    BN_free(zero);
    return ret;
}

static int witness(BIGNUM *w, const BIGNUM *a, const BIGNUM *a1,
                   const BIGNUM *a1_odd, int k, BN_CTX *ctx,
                   BN_MONT_CTX *mont)
{
    if (!BN_mod_exp_mont(w, w, a1_odd, a, ctx, mont)) /* w := w^a1_odd mod a */
        return -1;
    if (BN_is_one(w))
        return 0;               /* probably prime */
    if (BN_cmp(w, a1) == 0)
        return 0;               /* w == -1 (mod a), 'a' is probably prime */
    while (--k) {
        if (!BN_mod_mul(w, w, w, a, ctx)) /* w := w^2 mod a */
            return -1;
        if (BN_is_one(w))
            return 1;           /* 'a' is composite, otherwise a previous 'w'
                                 * would have been == -1 (mod 'a') */
        if (BN_cmp(w, a1) == 0)
            return 0;           /* w == -1 (mod a), 'a' is probably prime */
    }
    /*
     * If we get here, 'w' is the (a-1)/2-th power of the original 'w', and
     * it is neither -1 nor +1 -- so 'a' cannot be prime
     */
    bn_check_top(w);
    return 1;
}
```

Bibliography

[1] Yasemin Acar, Michael Backes, Sascha Fahl, Simson L. Garfinkel, Doowon Kim, Michelle L. Mazurek, and Christian Stransky. Comparing the usability of cryptographic APIs. In *2017 IEEE Symposium on Security and Privacy*, pages 154–171. IEEE Computer Society Press, May 2017.

[2] Yasemin Acar, Sascha Fahl, and Michelle L. Mazurek. You are not your developer, either: A research agenda for usable security and privacy research beyond end users. In *IEEE Cybersecurity Development, SecDev 2016, Boston, MA, USA, November 3-4, 2016*, pages 3–8. IEEE Computer Society, 2016.

[3] David Adrian, Karthikeyan Bhargavan, Zakir Durumeric, Pierrick Gaudry, Matthew Green, J. Alex Halderman, Nadia Heninger, Drew Springall, Emmanuel Thomé, Luke Valenta, Benjamin VanderSloot, Eric Wustrow, Santiago Zanella-Béguelin, and Paul Zimmermann. Imperfect forward secrecy: How Diffie-Hellman fails in practice. In Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors, *ACM CCS 2015*, pages 5–17. ACM Press, October 2015.

[4] Manindra Agrawal, Neeraj Kayal, and Nitin Saxena. PRIMES is in P. *Annals of mathematics*, pages 781–793, 2004.

[5] Martin R. Albrecht, Jake Massimo, Kenneth G. Paterson, and Juraj Somorovsky. Prime and prejudice: Primality testing under adversarial conditions. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, Toronto, Canada, October 15-19, 2018*, pages 281–298, 2018.

[6] Apple Inc. Common crypto - Security Apple Developer. `https://opensource.apple.com//source/CommonCrypto/`, August 2018.

[7] Apple Inc. corecrypto - Security Apple Developer. `https://developer.apple.com/security/`, August 2018.

[8] François Arnault. Constructing Carmichael numbers which are strong pseudoprimes to several bases. *Journal of Symbolic Computation*, 20(2):151–161, 1995.

[9] François Arnault. The Rabin-Monier theorem for Lucas pseudoprimes. *Mathematics of Computation of the American Mathematical Society*, 66(218):869–881, 1997.

[10] A Oliver L Atkin and François Morain. Elliptic curves and primality proving. *Mathematics of computation*, 61(203):29–68, 1993.

[11] Jane Austen. Pride and Prejudice. 1813.

[12] Nimrod Aviram, Sebastian Schinzel, Juraj Somorovsky, Nadia Heninger, Maik Dankel, Jens Steube, Luke Valenta, David Adrian, J. Alex Halderman, Viktor Dukhovni, Emilia Käsper, Shaanan Cohney, Susanne Engels, Christof Paar, and Yuval Shavitt. DROWN: breaking TLS using SSLv2. In Thorsten Holz and Stefan Savage, editors, *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016.*, pages 689–706. USENIX Association, 2016.

[13] Robert Baillie. OEIS A217120: Lucas pseudoprimes. `https://oeis.org/A217120`, March 2013.

[14] Robert Baillie. OEIS A217255: Strong Lucas pseudoprimes. `https://oeis.org/A217255`, March 2013.

[15] Robert Baillie and Samuel S Wagstaff. Lucas pseudoprimes. *Mathematics of Computation*, 35(152):1391–1417, 1980.

[16] Elaine Barker. NIST Special Publication 800–57 Part 1, Revision 4. *NIST, Tech. Rep*, 2016.

[17] Pierre Beauchemin, Gilles Brassard, Claude Crépeau, Claude Goutier, and Carl Pomerance. The generation of random numbers that are probably prime. *Journal of Cryptology*, 1(1):53–64, 1988.

[18] Mihir Bellare, Kenneth G. Paterson, and Phillip Rogaway. Security of symmetric encryption against mass surveillance. In Juan A. Garay and Rosario Gennaro, editors, *CRYPTO 2014, Part I*, volume 8616 of *LNCS*, pages 1–19. Springer, Heidelberg, August 2014.

[19] Daniel J. Bernstein. Curve25519: New Diffie-Hellman speed records. In Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin, editors, *PKC 2006*, volume 3958 of *LNCS*, pages 207–228. Springer, Heidelberg, April 2006.

[20] Daniel J. Bernstein, Tung Chou, Chitchanok Chuengsatiansup, Andreas Hülsing, Eran Lambooij, Tanja Lange, Ruben Niederhagen, and Christine van Vredendaal. How to manipulate curve standards: A white paper for the black hat `http://bada55.cr.yp.to`. In Liqun Chen and Shin'ichiro Matsuo, editors, *Security Standardisation Research - Second International Conference, SSR 2015, Tokyo, Japan, December 15-16, 2015, Proceedings*, volume 9497 of *Lecture Notes in Computer Science*, pages 109–139. Springer, 2015.

[21] Daniel J. Bernstein and Tanja Lange. Faster addition and doubling on elliptic curves. In Kaoru Kurosawa, editor, *ASIACRYPT 2007*, volume 4833 of *LNCS*, pages 29–50. Springer, Heidelberg, December 2007.

[22] Daniel J. Bernstein, Tanja Lange, and Peter Schwabe. The security impact of a new cryptographic library. In Alejandro Hevia and Gregory Neven, editors, *LATINCRYPT 2012*, volume 7533 of *LNCS*, pages 159–176. Springer, Heidelberg, October 2012.

[23] Benjamin Beurdouche, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, and Jean Karim Zinzindohoue. A messy state of the union: Taming the composite state machines of TLS. In *2015 IEEE Symposium on Security and Privacy*, pages 535–552. IEEE Computer Society Press, May 2015.

[24] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Alfredo Pironti, and Pierre-Yves Strub. Triple handshakes and cookie cutters: Breaking and fixing authentication over TLS. In *2014 IEEE Symposium on Security and Privacy*, pages 98–113. IEEE Computer Society Press, May 2014.

[25] S. Blake-Wilson, N. Bolyard, V. Gupta, C. Hawk, and B. Moeller. Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS).

RFC 4492 (Informational), May 2006. Obsoleted by RFC 8422, updated by RFCs 5246, 7027, 7919.

[26] Daniel Bleichenbacher. Breaking a cryptographic protocol with pseudoprimes. In Serge Vaudenay, editor, *PKC 2005*, volume 3386 of *LNCS*, pages 9–15. Springer, Heidelberg, January 2005.

[27] Hanno Böck, Aaron Zauner, Sean Devlin, Juraj Somorovsky, and Philipp Jovanovic. Nonce-disrespecting adversaries: Practical forgery attacks on GCM in TLS. In Natalie Silvanovich and Patrick Traynor, editors, *10th USENIX Workshop on Offensive Technologies, WOOT 16, Austin, TX, USA, August 8-9, 2016*. USENIX Association, 2016.

[28] Joppe W. Bos, Craig Costello, Patrick Longa, and Michael Naehrig. Selecting elliptic curves for cryptography: an efficiency and security analysis. *J. Cryptographic Engineering*, 6(4):259–286, 2016.

[29] Wieb Bosma, John Cannon, and Catherine Playoust. The Magma algebra system. *J. Symbolic Comput.*, 24, 1997. Computational algebra and number theory (London, 1993).

[30] Colin Boyd, Britta Hale, Stig Frode Mjølsnes, and Douglas Stebila. From stateless to stateful: Generic authentication and authenticated encryption constructions with application to TLS. In Kazue Sako, editor, *CT-RSA 2016*, volume 9610 of *LNCS*, pages 55–71. Springer, Heidelberg, February / March 2016.

[31] Jørgen Brandt and Ivan Damgård. On generation of probable primes by incremental search. In *Proceedings of the 12th Annual International Cryptology Conference on Advances in Cryptology*, pages 358–370. Springer-Verlag, 1992.

[32] Reinier Bröker and Peter Stevenhagen. Constructing elliptic curves in almost polynomial time. arXiv:math/0511729, 2005.

[33] Stephen Checkoway, Jacob Maskiewicz, Christina Garman, Joshua Fried, Shaanan Cohney, Matthew Green, Nadia Heninger, Ralf-Philipp Weinmann, Eric Rescorla, and Hovav Shacham. A systematic analysis of the juniper dual EC incident. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 468–479. ACM Press, October 2016.

[34] Stephen Checkoway, Ruben Niederhagen, Adam Everspaugh, Matthew Green, Tanja Lange, Thomas Ristenpart, Daniel J. Bernstein, Jake Maskiewicz, Hovav Shacham, and Matthew Fredrikson. On the practical exploitability of dual EC in TLS implementations. In Kevin Fu and Jaeyeon Jung, editors, *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014.*, pages 319–335. USENIX Association, 2014.

[35] Henri Cohen and Hendrik W Lenstra. Primality testing and jacobi sums. *Mathematics of computation*, 42(165):297–330, 1984.

[36] Don Coppersmith. Fast evaluation of logarithms in fields of characteristic two. *IEEE transactions on information theory*, 30(4):587–594, 1984.

[37] Giuseppe Cornacchia. Su di un metodo per la risoluzione in numeri interi dellequazione $\sum_{h=0}^{n} C_h x^{n-h} y^h = P$. *Giornale di Matematiche di Battaglini*, 46:33–90, 1908.

[38] Oracle Corporation. *OpenJDK 10 Open Java Development Kit*, 2018. `openjdk.java.net`.

[39] Richard Crandall and Carl Pomerance. *Prime Numbers: A Computational Perspective*, volume 182. Springer Science & Business Media, 2006. pp.136-140.

[40] Wei Dai. Crypto++. `https://www.cryptopp.com/`, April 2018.

[41] Ivan Damgård, Peter Landrock, and Carl Pomerance. Average case error estimates for the strong probable prime test. *Mathematics of Computation*, 61(203):177–194, 1993.

[42] Jean Paul Degabriele, Kenneth G. Paterson, Jacob C. N. Schuldt, and Joanne Woodage. Backdoors in pseudorandom number generators: Possibility and impossibility results. In Matthew Robshaw and Jonathan Katz, editors, *CRYPTO 2016, Part I*, volume 9814 of *LNCS*, pages 403–432. Springer, Heidelberg, August 2016.

[43] Bert Den Boer. Diffie-Hellman is as strong as discrete log for certain primes. In *Conference on the Theory and Application of Cryptography*, pages 530–539. Springer, 1988.

[44] Tom St Denis. LibTomCrypt. `http://www.libtom.net/LibTomCrypt/`, April 2018.

[45] Tom St Denis. LibTomMath. `http://www.libtom.net/LibTomMath/`, April 2018.

[46] Tom St Denis. TomsFastMath. `http://www.libtom.net/TomsFastMath/`, April 2018.

[47] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard), August 2008. Obsoleted by RFC 8446, updated by RFCs 5746, 5878, 6176, 7465, 7507, 7568, 7627, 7685, 7905, 7919, 8447.

[48] Whitfield Diffie and Martin Hellman. New directions in cryptography. *IEEE transactions on Information Theory*, 22(6):644–654, 1976.

[49] Yevgeniy Dodis, Chaya Ganesh, Alexander Golovnev, Ari Juels, and Thomas Ristenpart. A formal treatment of backdoored pseudorandom generators. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part I*, volume 9056 of *LNCS*, pages 101–126. Springer, Heidelberg, April 2015.

[50] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. An empirical study of cryptographic misuse in Android applications. In Sadeghi et al. [141], pages 73–84.

[51] P. Erdös. On pseudoprimes and Carmichael numbers. *Publ. Math. Debrecen*, 4:201–206, 1956.

[52] Sascha Fahl, Marian Harbach, Henning Perl, Markus Koetter, and Matthew Smith. Rethinking SSL development in an appified world. In Sadeghi et al. [141], pages 49–60.

[53] Marc Fischlin, Felix Günther, Giorgia Azzurra Marson, and Kenneth G. Paterson. Data is a stream: Security of stream-based channels. In Rosario Gennaro and Matthew J. B. Robshaw, editors, *CRYPTO 2015, Part II*, volume 9216 of *LNCS*, pages 545–564. Springer, Heidelberg, August 2015.

[54] Joshua Fried, Pierrick Gaudry, Nadia Heninger, and Emmanuel Thomé. A kilobit hidden SNFS discrete logarithm computation. In Jean-Sébastien Coron

and Jesper Buus Nielsen, editors, *EUROCRYPT 2017, Part I*, volume 10210 of *LNCS*, pages 202–231. Springer, Heidelberg, April / May 2017.

[55] Bundesamt für Sicherheit in der Informationstechnik. BSI TR-02102-1 Cryptographic Mechanisms: Recommendations and Key Lengths. Technical guideline, Federal Office for Information Security, January 2017.

[56] Steven D Galbraith. *Mathematics of Public Key Cryptography*. Cambridge University Press, 2012.

[57] Steven D. Galbraith, Jake Massimo, and Kenneth G. Paterson. Safety in numbers: On the need for robust Diffie-Hellman parameter validation. In Dongdai Lin and Kazue Sako, editors, *PKC 2019, Part II*, volume 11443 of *LNCS*, pages 379–407. Springer, Heidelberg, April 2019.

[58] Joseph Gallian. *Contemporary Abstract Algebra, the Fundamental Theorem of Cyclic Groups*. Nelson Education, 2012.

[59] Carl Friedrich Gauss and Johann Carl Friedrich Gauss. *Disquisitiones Arithmeticae*, volume 157. Yale University Press, 1966.

[60] Jeff Gilchrist. Pseudoprime enumeration with probabilistic primality tests. `http://gilchrist.ca/jeff/factoring/pseudoprimes.html`, August 2013.

[61] D. Gillmor. Negotiated Finite Field Diffie-Hellman Ephemeral Parameters for Transport Layer Security (TLS). RFC 7919 (Proposed Standard), August 2016.

[62] Google. The Go Programming Language. `https://golang.org`, July 2018.

[63] Daniel M. Gordon. Designing and detecting trapdoors for discrete log cryptosystems. In Ernest F. Brickell, editor, *CRYPTO'92*, volume 740 of *LNCS*, pages 66–75. Springer, Heidelberg, August 1993.

[64] Daniel M Gordon. Discrete logarithms in gf(p) using the number field sieve. *SIAM Journal on Discrete Mathematics*, 6(1):124–138, 1993.

[65] Peter Leo Gorski, Luigi Lo Iacono, Dominik Wermke, Christian Stransky, Sebastian Möller, Yasemin Acar, and Sascha Fahl. Developers deserve security warnings, too: On the effect of integrated security advice on cryptographic API misuse. In Mary Ellen Zurko and Heather Richter Lipford, editors, *Fourteenth*

Symposium on Usable Privacy and Security, SOUPS 2018, Baltimore, MD, USA, August 12-14, 2018., pages 265–281. USENIX Association, 2018.

[66] Torbjorn Granlund and the GMP development team. GNU MP: The GNU Multiple Precision Arithmetic Library. `https://gmplib.org`, April 2018.

[67] Andrew Granville and Carl Pomerance. Two contradictory conjectures concerning Carmichael numbers. *Mathematics of Computation*, 71(238):883–908, 2002.

[68] Matthew Green and Matthew Smith. Developers are not the enemy!: The need for usable security APIs. *IEEE Security & Privacy*, 14(5):40–46, 2016.

[69] Shay Gueron, Adam Langley, and Yehuda Lindell. AES-GCM-SIV: specification and analysis. *IACR Cryptology ePrint Archive*, 2017:168, 2017.

[70] Peter Gutmann. Lessons learned in implementing and deploying crypto software. In Dan Boneh, editor, *Proceedings of the 11th USENIX Security Symposium, San Francisco, CA, USA, August 5-9, 2002*, pages 315–325. USENIX, 2002.

[71] Peter Gutmann. CryptLib. `http://www.cryptlib.com/`, April 2018.

[72] F. Hao (Ed.). J-PAKE: Password-Authenticated Key Exchange by Juggling. RFC 8236 (Informational), September 2017.

[73] Nadia Heninger, Zakir Durumeric, Eric Wustrow, and J. Alex Halderman. Mining your ps and qs: Detection of widespread weak keys in network devices. In Tadayoshi Kohno, editor, *USENIX Security 2012*, pages 205–220. USENIX Association, August 2012.

[74] Jeffrey Hoffstein, Jill Pipher, and Joseph H Silverman. *An Introduction to Mathematical Cryptography*, volume 1. Springer, 2008.

[75] Andreas Höglund. MPZ_SPSP's under GMP 5.0.1. `http://www.hoegge.dk/gmp/gmp501.htm`, 2016. Last accessed 2016-10-31.

[76] Dana Jacobsen. Pseudoprime Statistics, Tables, and Data. `http://ntheory.org/pseudoprimes.html`, August 2015.

[77] Gerhard Jaeschke. On strong pseudoprimes to several bases. *Mathematics of Computation*, 61(204):915–926, 1993.

[78] David Jao and Luca De Feo. Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies. In Bo-Yin Yang, editor, *Post-Quantum Cryptography - 4th International Workshop, PQCrypto 2011*, pages 19–34. Springer, Heidelberg, November / December 2011.

[79] Antoine Joux and Reynald Lercier. Improvements to the general number field sieve for discrete logarithms in prime fields. A comparison with the gaussian integer method. *Mathematics of computation*, 72(242):953–967, 2003.

[80] Marc Joye and Pascal Paillier. Fast generation of prime numbers on portable devices: An update. In Louis Goubin and Mitsuru Matsui, editors, *CHES 2006*, volume 4249 of *LNCS*, pages 160–173. Springer, Heidelberg, October 2006.

[81] Marc Joye, Pascal Paillier, and Serge Vaudenay. Efficient generation of prime numbers. In Çetin Kaya Koç and Christof Paar, editors, *CHES 2000*, volume 1965 of *LNCS*, pages 340–354. Springer, Heidelberg, August 2000.

[82] Achim Jung. Implementing the RSA cryptosystem. *Computers & Security*, 6(4):342–350, 1987.

[83] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography*. Chapman and Hall/CRC, 2014.

[84] Cameron F Kerry and Patrick D Gallagher. FIPS PUB 186–4 federal information processing standards publication digital signature standard (DSS). *National Institute of Standards and Technology*, 2013.

[85] Donald E Knuth. *Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley Professional, 2014.

[86] Werner Koch. Github - Libgcrypt changes to default primality test. `https://github.com/gpg/libgcrypt/commit/78a84338cb36748f17cc444b17ab7033ce384c34#diff-96a06fc4d0080caec00d423ca08a6c86`, April 2005.

[87] Werner Koch. Libgcrypt. `https://gnupg.org/software/libgcrypt/index.html`, April 2018.

[88] Alwin Korselt. Probleme chinois. *Lintermédiaire math*, 6:143–143, 1899.

[89] David Lazar, Haogang Chen, Xi Wang, and Nickolai Zeldovich. Why does cryptographic software fail?: A case study and open problems. In *Asia-Pacific Workshop on Systems, APSys'14, Beijing, China, June 25-26, 2014*, pages 7:1–7:7. ACM, 2014.

[90] M. Lepinski and S. Kent. Additional Diffie-Hellman Groups for Use with IETF Standards. RFC 5114 (Informational), January 2008.

[91] LibTomMath. Pull request - added FIPS 186.4 compliance, an additional strong Lucas-Selfridge (for BPSW). `https://github.com/libtom/libtommath/pull/113`, August 2018.

[92] Chae Hoon Lim and Pil Joong Lee. A key recovery attack on discrete log-based schemes using a prime order subgroup. In Burton S. Kaliski Jr., editor, *CRYPTO'97*, volume 1294 of *LNCS*, pages 249–263. Springer, Heidelberg, August 1997.

[93] Telegram FZ LLC. Telegram Messenger. `https://telegram.org`, 2018.

[94] Jack Lloyd. Botan. `https://github.com/randombit/botan`, August 2018.

[95] Jack Lloyd. Botan pull request - add Lucas test from FIPS 186-4. `https://github.com/randombit/botan/pull/1636`, August 2018.

[96] Jack Lloyd. Botan github repository. `https://github.com/randombit/botan/blob/5d74496ee51b8a2d1c418b0a66bddac6f0263749/src/lib/math/numbertheory/primality.cpp\#L51`, August 2020.

[97] Macsyma group. *Maxima 5.41.0, Documentation*, 2017. Available at `http://maxima.sourceforge.net/docs/manual/maxima_10.html`.

[98] Macsyma group. *Maxima 5.41.0, a Computer Algebra System*, 2018. Available at `http://maxima.sourceforge.net/index.html`.

[99] Marcel Martin. *PRIMO–Primality Proving*, 2016. `https://www.ellipsa.eu`.

[100] Jake Massimo and Kenneth G. Paterson. A performant, misuse-resistant API for primality testing. Cryptology ePrint Archive, Report 2020/065, 2020. `https://eprint.iacr.org/2020/065`.

[101] Ueli M. Maurer. Towards the equivalence of breaking the Diffie-Hellman protocol and computing discrete algorithms. In Yvo Desmedt, editor, *CRYPTO'94*, volume 839 of *LNCS*, pages 271–281. Springer, Heidelberg, August 1994.

[102] Ueli M. Maurer. Fast generation of prime numbers and secure public-key cryptographic parameters. *Journal of Cryptology*, 8(3):123–155, September 1995.

[103] Ueli M. Maurer and Stefan Wolf. Diffie-Hellman oracles. In Neal Koblitz, editor, *CRYPTO'96*, volume 1109 of *LNCS*, pages 268–282. Springer, Heidelberg, August 1996.

[104] Jud McCranie. OEIS A014233: Smallest odd number for which Miller-Rabin primality test on bases less than or equal to the n-th prime does not reveal compositeness. `https://oeis.org/A014233`, Feb 1997.

[105] Alfred J Menezes, Paul C Van Oorschot, and Scott A Vanstone. *Handbook of Applied Cryptography*. CRC press, 1996.

[106] Gary L Miller. Riemann's hypothesis and tests for primality. In *Proceedings of seventh annual ACM symposium on Theory of computing*, pages 234–239. ACM, 1975.

[107] Ilya Mironov. Factoring RSA Moduli. Part II. `https://windowsontheory.org/2012/05/17/factoring-rsa-moduli-part-ii/`, May 2012.

[108] Louis Monier. Evaluation and comparison of two efficient probabilistic primality testing algorithms. *Theoretical Computer Science*, 12(1):97–108, 1980.

[109] Mozilla. Mozilla Network Security Services (NSS). https://developer.mozilla.org/en-US/docs/Mozilla/Projects/NSS, 2020.

[110] Mozilla. MPI arbitrary precision integer arithmetic library mozsearch. `https://searchfox.org/nss/rev/7a530724d8df48df8cbddb64adae172937e646d4/lib/freebl/mpi/mpi.h#72`, Feb 2020.

[111] Sarah Nadi, Stefan Krüger, Mira Mezini, and Eric Bodden. "Jumping Through Hoops": Why do Java developers struggle with cryptography APIs? In

Jan Jürjens and Kurt Schneider, editors, *Software Engineering 2017, Fachtagung des GI-Fachbereichs Softwaretechnik, 21.-24. Februar 2017, Hannover, Deutschland*, volume P-267 of *LNI*, page 57. GI, 2017.

[112] Alena Naiakshina, Anastasia Danilova, Christian Tiefenau, Marco Herzog, Sergej Dechand, and Matthew Smith. Why do developers get password storage wrong?: A qualitative usability study. In Thuraisingham et al. [155], pages 311–328.

[113] Shyam Narayanan. *Improving the Speed and Accuracy of the Miller-Rabin Primality Test*. MIT PRIMES-USA, 2014. `https://math.mit.edu/research/highschool/primes/materials/2014/Narayanan.pdf`.

[114] Matús Nemec, Marek Sýs, Petr Svenda, Dusan Klinec, and Vashek Matyas. The return of coppersmith's attack: Practical factorization of widely used RSA moduli. In Thuraisingham et al. [155], pages 1631–1648.

[115] Thomas Nicely. GNU GMP mpz_probab_prime_p pseudoprimes. `http://www.trnicely.net/misc/mpzspsp.html`, 2016. Last accessed 2016-10-31.

[116] Legion of the Bouncy Castle Inc. *The Bouncy Castle Crypto Package For C Sharp*, 2018. `https://github.com/bcgit/bc-csharp`.

[117] GitHub The OpenSSL Project. Pull request - Increase number of MR tests for RSA prime generation #6075. `https://github.com/openssl/openssl/pull/6075`, August 2018.

[118] The OpenSSL Project. OpenSSL 1.0.2o: The Open Source toolkit for SSL/TLS. `https://github.com/openssl/openssl/releases/tag/OpenSSL_1_0_2o`, Mar 2018.

[119] The OpenSSL Project. OpenSSL 1.0.2p: The Open Source toolkit for SSL/TLS. `https://github.com/openssl/openssl/releases/tag/OpenSSL_1_0_2p`, Aug 2018.

[120] The OpenSSL Project. OpenSSL 1.1.0h: The Open Source toolkit for SSL/TLS. `https://github.com/openssl/openssl/releases/tag/OpenSSL_1_1_0h`, Mar 2018.

[121] The OpenSSL Project. OpenSSL 1.1.0i: The Open Source toolkit for SSL/TLS. https://github.com/openssl/openssl/releases/tag/OpenSSL_1_1_0i, Aug 2018.

[122] The OpenSSL Project. OpenSSL 1.1.1-pre6: The Open Source toolkit for SSL/TLS. https://github.com/openssl/openssl/releases/tag/OpenSSL_1_1_1-pre6, May 2018.

[123] The OpenSSL Project. OpenSSL 1.1.1-pre9: The Open Source toolkit for SSL/TLS. https://github.com/openssl/openssl/releases/tag/OpenSSL_1_1_1-pre9, Aug 2018.

[124] The OpenSSL Project. OpenSSL: The Open Source toolkit for SSL/TLS. www.openssl.org, May 2018.

[125] The OpenSSL Project. OpenSSL 1.1.1c: The Open Source toolkit for SSL/TLS. https://github.com/openssl/openssl/releases/tag/OpenSSL_1_1_1c, May 2019.

[126] The OpenSSL Project. OpenSSL 1.1.1d: The Open Source toolkit for SSL/TLS. https://github.com/openssl/openssl/releases/tag/OpenSSL_1_1_1d, May 2019.

[127] Christopher Patton and Thomas Shrimpton. Partially specified channels: The TLS 1.3 record layer without elision. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 1415–1428. ACM Press, October 2018.

[128] Christophe Petit and Jean-Jacques Quisquater. On polynomial systems arising from a Weil descent. In Xiaoyun Wang and Kazue Sako, editors, *ASIACRYPT 2012*, volume 7658 of *LNCS*, pages 451–466. Springer, Heidelberg, December 2012.

[129] Richard GE Pinch. The Carmichael numbers up to 10^{21}. In *Proceedings Conference on Algorithmic Number Theory*, volume 46 of *Turku Centre for Computer Science General Publications*, 2008.

[130] Stephen Pohlig and Martin Hellman. An improved algorithm for computing logarithms over GF(p) and its cryptographic significance. *IEEE Transactions on information Theory*, 24(1):106–110, 1978.

[131] JM Pollard. A Monte Carlo method for factorization. *BIT Numerical Mathematics*, 15(3):331–334, 1975.

[132] John M Pollard. Monte carlo methods for index computation (mod p). *Mathematics of computation*, 32(143):918–924, 1978.

[133] Carl Pomerance. Are there counter-examples to the Baillie-PSW primality test. Dopo Le Parole aangeboden aan Dr. A. K. Lenstra., 1984.

[134] Carl Pomerance, John L Selfridge, and Samuel S Wagstaff. The pseudoprimes to $25 \cdot 10^9$. *Mathematics of Computation*, 35(151):1003–1026, 1980.

[135] Michael O Rabin. Probabilistic algorithm for testing primality. *Journal of number theory*, 12(1):128–138, 1980.

[136] Wolfram Research, Inc. Mathematica, Version 11.3, 2018. Champaign, IL, 2018.

[137] Gerhard Rieger. *Socat security advisory 7 - Openwall oss-security mailing list*, 2016. `http://www.openwall.com/lists/oss-security/2016/02/01/4`.

[138] Phillip Rogaway. Nonce-based symmetric encryption. In Bimal K. Roy and Willi Meier, editors, *FSE 2004*, volume 3017 of *LNCS*, pages 348–359. Springer, Heidelberg, February 2004.

[139] Phillip Rogaway and Thomas Shrimpton. A provable-security treatment of the key-wrap problem. In Serge Vaudenay, editor, *EUROCRYPT 2006*, volume 4004 of *LNCS*, pages 373–390. Springer, Heidelberg, May / June 2006.

[140] J Barkley Rosser, Lowell Schoenfeld, et al. Approximate formulas for some functions of prime numbers. *Illinois Journal of Mathematics*, 6(1):64–94, 1962.

[141] Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors. *ACM CCS 2013*. ACM Press, November 2013.

[142] Oliver Schirokauer. Virtual logarithms. *Journal of Algorithms*, 57(2):140–147, 2005.

[143] Atle Selberg. An elementary proof of the prime-number theorem. *Annals of Mathematics*, pages 305–313, 1949.

[144] Daniel Shanks. Class number, a theory of factorization, and genera. In *Proc. of Symp. Math. Soc., 1971*, volume 20, pages 41–440, 1971.

[145] Joseph H Silverman. *A Friendly Introduction To Number Theory*, volume 3. Prentice Hall, Upper Saddle River, NJ, 2006.

[146] British Standards. ISO/IEC 18032:2005 Information technology – Security techniques – Prime number generation. Standard, International Organization for Standardization, January 2005.

[147] British Standards. ISO/IEC DIS 18032 Information technology – Security techniques – Prime number generation. Standard, International Organization for Standardization, 2019.

[148] William Stein et al. *Sage Mathematics Software Version 8.2*. The Sage Development Team, 2017. Available at `http://www.sagemath.org`.

[149] Falko Strenzke. An analysis of OpenSSL's random number generator. In Marc Fischlin and Jean-Sébastien Coron, editors, *EUROCRYPT 2016, Part I*, volume 9665 of *LNCS*, pages 644–669. Springer, Heidelberg, May 2016.

[150] SymPy. SymPy GitHub repository. Available at `https://github.com/sympy/sympy/commit/9e35a94eceaff73b350794dcc70b4a412dc2f6e6#diff-e20bc128d13486b598a04fce77584900`, 2017.

[151] SymPy Development Team. *SymPy: Python library for symbolic mathematics*, 2017.

[152] D. Taylor, T. Wu, N. Mavrogiannopoulos, and T. Perrin. Using the Secure Remote Password (SRP) Protocol for TLS Authentication. RFC 5054 (Informational), November 2007.

[153] The PARI Group, Univ. Bordeaux. *PARI/GP Frequently Asked Questions*, 2018. Available from `http://pari.math.u-bordeaux.fr/faq.html#primetest`.

[154] The PARI Group, Univ. Bordeaux. *PARI/GP version* `2.9.0`, 2018. Available from `http://pari.math.u-bordeaux.fr/`.

[155] Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors. *ACM CCS 2017*. ACM Press, October / November 2017.

[156] Luke Valenta, David Adrian, Antonio Sanso, Shaanan Cohney, Joshua Fried, Marcella Hastings, J. Alex Halderman, and Nadia Heninger. Measuring small subgroup attacks against Diffie-Hellman. In *NDSS 2017*. The Internet Society, February / March 2017.

[157] Waterloo Maple (Maplesoft). *Maple Version 2017*, 2017. Available at `https://www.maplesoft.com/products/Maple/`.

[158] Eric W. Weisstein. Baillie-PSW primality test from MathWorld-a Wolfram web resource. `http://mathworld.wolfram.com/Baillie-PSWPrimalityTest.html`, April 2018.

[159] WolfSSL Inc. Pull request - Prime Number Testing. `https://github.com/wolfSSL/wolfssl/pull/1665`, August 2018.

[160] WolfSSL Inc. WolfSSL. `https://www.wolfssl.com/wolfSSL/Home.html`, April 2018.

[161] David Wong. How to Backdoor Diffie-Hellman. Cryptology ePrint Archive, Report 2016/644, 2016. `https://eprint.iacr.org/2016/644`.

[162] T. Wu. The SRP Authentication and Key Exchange System. RFC 2945 (Proposed Standard), September 2000.

[163] Tom Wu. JSBN: RSA and ECC in JavaScript. `http://www-cs-students.stanford.edu/~tjw/jsbn/`, April 2017.

[164] Glenn Wurster and Paul C. van Oorschot. The developer is the enemy. In Matt Bishop, Christian W. Probst, Angelos D. Keromytis, and Anil Somayaji, editors, *Proceedings of the 2008 Workshop on New Security Paradigms, Lake Tahoe, CA, USA, September 22-25, 2008*, pages 89–97. ACM, 2008.

[165] Scott Yilek, Eric Rescorla, Hovav Shacham, Brandon Enright, and Stefan Savage. When private keys are public: Results from the 2008 debian OpenSSL vulnerability. In *Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement*, pages 15–27. ACM, 2009.

[166] Adam Young and Moti Yung. Kleptography: Using cryptography against cryptography. In Walter Fumy, editor, *EUROCRYPT'97*, volume 1233 of *LNCS*, pages 62–74. Springer, Heidelberg, May 1997.